

2

AD-A216 644

LOG(F): AN OPTIMAL COMBINATION OF LOGIC PROGRAMMING, REWRITING,
AND LAZY EVALUATION

DTIC
ELECTE
JAN 19 1990
S D & D

Sanjai Narain

April 1988

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

P-7437

90 01 16 026

The RAND Corporation

Papers are issued by The RAND Corporation as a service to its professional staff. Their purpose is to facilitate the exchange of ideas among those who share the author's research interests; Papers are not reports prepared in fulfillment of RAND's contracts or grants. Views expressed in a Paper are the author's own and are not necessarily shared by RAND or its research sponsors.

The RAND Corporation, 1700 Main Street, P.O. Box 2138, Santa Monica, CA 90406-2138

LOG(F): An Optimal Combination of Logic Programming, Rewriting, and Lazy Evaluation

Sanjai Narain
Rand Corporation
1700 Main Street
Santa Monica, CA 90406

ABSTRACT

A new approach for combining logic programming, rewriting, and lazy evaluation is described. It rests upon *subsuming* within logic programming, instead of upon *extending* it with, rewriting, and lazy evaluation.

A non-terminating, non-deterministic rewrite rule system, F^* and a reduction strategy for it, *select*, are defined. F^* is shown to be reduction-complete in that *select* simplifies terms whenever possible. A class of F^* programs called Deterministic F^* is defined and shown to satisfy confluence, directedness, and minimality. Confluence ensures that every term can be simplified in at most one way. Directedness eliminates searching in simplification of terms. Minimality ensures that *select* simplifies terms in a minimum number of steps. Completeness and minimality enable *select* to exhibit, respectively, weak and strong forms of laziness.

F^* can be compiled into Horn clauses in such a way that when SLD-resolution interprets these, it directly simulates the behavior of *select*. Thus, SLD-resolution is made to exhibit laziness. LOG(F) is defined to be a logic programming system augmented with an F^* compiler, and the equality axiom $X=X$. LOG(F) can be used to do lazy functional programming in logic, implement useful cases of the rule of substitution of equals for equals, and obtain a new proof of confluence for combinatory logic. (RW)



Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By <i>etc. on file</i>	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

Table of Contents

I. Introduction	I-1
I.1 The problem.....	I-1
I.2 Summary of main results.....	I-2
I.2.1 A rewrite rule system F^*	I-2
I.2.2 Deterministic F^*	I-3
I.2.3 Compiling F^* into Horn clauses	I-4
I.2.4 $\text{LOG}(F)$	I-5
I.2.5 Applications of $\text{LOG}(F)$	I-6
I.3 Relationship with previous work	I-7
I.4 Outline of paper	I-10
II. A rewrite rule system F^*	II-1
II.1 Introduction	II-1
II.2 Definition of F^*	II-1
II.3 Reduction-completeness of F^*	II-7
III. Deterministic F^*	III-1
III.1 Introduction.....	III-1
III.2 Definition of DF^*	III-1
III.3 Confluence and directedness of DF^*	III-3
IV. Labeled deterministic F^*	IV-1
IV.1 Introduction.....	IV-1
IV.2 Definition of LDF^*	IV-2
IV.3 Minimality of LDF^*	IV-5
IV.3.1 Existence of successful leftmost NA-derivations.....	IV-6
IV.3.2 E-sets of N-reductions.....	IV-8
IV.3.3 Reductions of proper terms	IV-8
IV.4 Extension of minimality result to normal forms	IV-11
IV.5 Derived minimality of DF^*	IV-11
V. Compilation of F^* into Horn clauses	V-1
V.1 Introduction.....	V-1
V.2 Compilation algorithm	V-1
V.3 Computing and printing normal forms.....	V-3
V.4 Optimizing rules satisfying restriction (g)	V-3

V.5 Computing functions eagerly in F*	V-4
V.6 Compiling LDF* programs	V-6
V.7 Correctness of F* compilation algorithm	V-7
VI. Programming in LOG(F)	VI-1
VI.1 Introduction	VI-1
VI.2 Non-determinism in F*	VI-1
VI.3 Implementing substitution of equals for equals	VI-3
VI.4 Combinatory logic	VI-4
VI.5 Two way communication	VI-5
VI.6 Hamming's problem	VI-6
VI.7 Infinite graphical structures	VI-7
VI.8 First two elements of a list	VI-11
VI.9 Comparing LOG(F) performance with that of Prolog	VI-12
VII. Summary and conclusions	VII-1
VIII. Acknowledgements	VIII-1
IX. References	IX-1

I. INTRODUCTION

I.1 THE PROBLEM

Logic programming [Kowalski 1979], is the use of statements of logic as computer programs. It has led to new insights into computing as well as logic. Rewriting is synonymous with reduction, as described, for example, in [Knuth & Bendix 1970]. It is simplification of an expression by successive application of some collection of rewrite rules. Its usefulness is evident from its appearance in many branches of mathematics. Lazy evaluation, e.g. [Vuillemin 1974], is a method of computing which ensures that a computation step is performed only when there is need to perform it. Thus, not only does it enable certain computations to terminate more quickly, it also enables computation with infinite data structures.

A system in which logic programming, rewriting, and lazy evaluation were combined could put considerable programming power at our disposal. In particular, it would simultaneously afford the expressive power of both functions, and relations.

Furthermore, such a system could be used to implement instances of the rule of substitution of equals for equals in logical statements. This is a very important rule, as witness its use in the simplest of mathematical derivations, e.g. solution of trigonometric identities. Logical statements could be expressed using logic programs, while equality theories could be expressed using rewrite rules.

We propose a new approach for building the above system which is rigorous, as well as computationally efficient. It rests upon *subsuming* within logic programming, instead of *extending* it with, rewriting and lazy evaluation. This means that SLD-resolution, the proof procedure used for logic programming, is not changed. Instead, Horn clauses, or *pure* Prolog clauses are written in such a way, that when SLD-resolution interprets them, it directly simulates lazy rewriting. The resulting system is called LOG(F). It can be said to make contributions to the following three areas:

- 1. Rewriting.** Simple, syntactic conditions are defined under which non-terminating, non-deterministic rewrite rules, with pattern matching, satisfy useful computational properties. These are regarding demand-driven reduction, confluence, elimination of search during reduction, and lengths of reductions.
- 2. Combination of logic programming and rewriting.** It is shown how rewriting can be *subsumed* within logic programming. In particular, the above computational properties of rewriting are realized within logic programming, without changing it, and without sacrificing logical rigor. This has two important

consequences.

First, a satisfactory combination of logic programming, and rewriting is achieved, *without* developing a new computational model of which the two are instances. Developing such a model is quite difficult, particularly if it is to have satisfactory declarative, *and* satisfactory procedural semantics. Second, full advantage is taken of the very efficient implementations of Prologs. Thus, formidable problems that implementation of the new model would very likely pose, are avoided.

3. Lazy evaluation. By 1 and 2, it is shown how lazy evaluation can be done *efficiently, within* the normally eager framework of logic programming. Thus a basis is established for understanding lazy evaluation purely in terms of well understood ideas in first order logic. Also, a new, and powerful use is found for an old, and widely used tool, namely, Prolog.

I.2 SUMMARY OF MAIN RESULTS

I.2.1 A rewrite rule system F^*

A first-order rewrite rule system F^* , with pattern matching, is defined. The function symbols are partitioned *in advance* into constructors, and non-constructors.

Simplification in F^* means reducing ground terms to simplified forms i.e. terms of the form $c(t_1, \dots, t_n)$ where c is a constructor symbol, and each of t_1, \dots, t_n is a ground term.

Simplified forms can be used to represent finite approximations to infinite structures, and are analogous to head-normal forms in the lambda-calculus [Wadsworth 1976]. In contrast, a normal form is defined to be a term in which all function symbols are constructors.

Now, an important point is that a method for computing simplified forms can be used repeatedly to compute normal forms. Moreover, it would terminate more often than would a method which directly computes normal forms. Hence it is sufficient to develop, and study properties of, a method for computing simplified forms.

An F^* program is a finite set of rules, each of the form $LHS \Rightarrow RHS$, satisfying the following restrictions: (1) LHS is of the form $f(L_1, \dots, L_m)$, $m \geq 0$, f a non-constructor function symbol, and each L_i either a variable, or of the form $c(T_1, \dots, T_n)$, $n \geq 0$, c a constructor symbol, and each T_i a variable, (2) a variable occurs at most once in LHS, and (3) all variables of RHS occur in LHS. Note that *non-terminating, non-deterministic* sets of rewrite rules are permissible. Also any rule with left hand side of depth greater than two can easily be expressed in terms of rules with left hand sides of depth at most

two, as required by (1).

Where P is an F^* program, and $f(T_1, \dots, T_n)$ a ground term, a reduction strategy for P , select_P , is defined by the following pseudo-Horn clauses:

$$\begin{aligned} &\text{select}_P(f(T_1, \dots, T_n), f(T_1, \dots, T_n)) \text{ if } f(T_1, \dots, T_n) \Rightarrow_P X. \\ &\text{select}_P(f(T_1, \dots, T_i, \dots, T_n), X) \text{ if} \\ &\quad \text{there is a rule } f(L_1, \dots, L_i, \dots, L_n) \Rightarrow \text{RHS in } P, \text{ and} \\ &\quad \text{there is no substitution } \sigma \text{ such that } T_i = L_i \sigma, \text{ and} \\ &\quad \text{select}_P(T_i, X). \end{aligned}$$

Here $A \Rightarrow B$ means there is a rule $\text{LHS} \Rightarrow \text{RHS}$ such that A matches LHS with substitution σ , and B is $\text{RHS}\sigma$. Select is shown to be *reduction-complete*, in that if a term can be simplified, it can be simplified by reducing it via select. Thus select exhibits a *weak form of laziness*. An example of an F^* program is:

$$\begin{aligned} &\text{perm}([]) \Rightarrow []. \\ &\text{perm}([A|V]) \Rightarrow \text{insert}(U, \text{perm}(V)). \\ &\text{insert}(U, X) \Rightarrow [U|X]. \\ &\text{insert}(U, [A|B]) \Rightarrow [A|\text{insert}(U, B)]. \end{aligned}$$

Here $[]$, $|$ are constructors, while perm, insert are non-constructors. The term $\text{perm}([1, 2, 3])$ is now reduced by select to $[1|\text{perm}([2, 3])]$, $[2|\text{insert}(1, \text{perm}([3]))]$, and $[3|\text{insert}(1, \text{insert}(2, \text{perm}([])))]$. If further reduction is desired, select may be called recursively upon the arguments of $|$ to yield each of $[1, 2, 3]$, $[1, 3, 2]$, $[2, 3, 1]$, $[2, 1, 3]$, $[3, 1, 2]$, $[3, 2, 1]$.

1.2.2 Deterministic F^*

An F^* program P is a DF^* program if (1) left hand sides of no two rules in P unify, and (2) where $f(L_1, \dots, L_i, \dots, L_m) \Rightarrow \text{RHS}$ is a rule in P , and L_i is not a variable, then in every other rule $f(K_1, \dots, K_i, \dots, K_m) \Rightarrow \text{RHS}_1$ in P , K_i is not a variable. These restrictions are very reasonable, and as examples throughout this paper show, it is possible to adhere to these, yet write quite expressive programs.

DF^* is shown to satisfy *confluence*, *directedness*, and *minimality*. Confluence ensures that a term can be simplified in at most one way. Directedness ensures that to simplify a term it is sufficient to compute *any* reduction computable by select. Moreover, all reductions computable by select are of equal length. Thus, during reduction, no searching is necessary. Provided, whenever a term is reduced, all copies of it are simultaneously reduced, minimality ensures that select simplifies terms in a minimum

number of steps. *Thus, select exhibits a strong form of laziness.* An example of a DF* program is:

```
append([],X)=>X.
append([U|V],W)=>[U|append(V,W)].
interleave([U|V],X)=>[U|interleave(X,V)].
a=>[1|a].
b=>[2|b].
```

However, the F* program above to insert an element non-deterministically into a list is not in DF*.

1.2.3 Compiling F* into Horn clauses

F* programs can be compiled into Horn clauses in such a way that when SLD-resolution interprets these, it directly simulates the behavior of (the interpreter based upon) select. This means that there is, essentially, a one-to-one correspondence between steps executed by (the interpreter based upon) select, and steps executed by SLD-resolution. This is accomplished by translating each F* rule into a distinct Horn clause, and *simultaneously* embodying in that clause, information about the logic of the rule, as well as information about the control of select when interpreting that rule.

Thus, SLD-resolution is made to exhibit laziness. If the F* program is also in DF*, clauses can be further transformed to eliminate all backtracking. Finally, clauses can be compiled into machine code by Prolog compilers. The compilation algorithm consists of two steps:

Step 1. For each n -ary, $n \geq 0$, constructor symbol c in P , and where X_1, \dots, X_n are distinct variables, generate the clause:

$$\text{reduce}(c(X_1, \dots, X_n), c(X_1, \dots, X_n))$$

Step 2. Let $f(L_1, \dots, L_m) \Rightarrow \text{RHS}$ be a rule in P . Let $A_1, \dots, A_m, \text{Out}$ be distinct Prolog variables not occurring in the rule. If L_i is a variable let Q_i be $A_i = L_i$. If L_i is $c(X_1, \dots, X_n)$ where c is a constructor symbol, and each X_i a variable, let Q_i be $\text{reduce}(A_i, c(X_1, \dots, X_n))$. Generate the clause:

$$\text{reduce}(f(A_1, \dots, A_m), \text{Out}) :- Q_1, \dots, Q_m, \text{reduce}(\text{RHS}, \text{Out}).$$

In practice, if L_i is a variable, Q_i can be dropped, provided A_i is replaced by L_i in $f(A_1, \dots, A_m)$. For example, the above F*, and DF* programs are compiled into:

```

reduce([],[]).
reduce([U|V],[U|V]).

reduce(perm(X),Z):-reduce(X,[],reduce([],Z).
reduce(perm(X),Z):-reduce(X,[FX|RX]),reduce(insert(FX,perm(RX)),Z).
reduce(insert(A,X),Z):-reduce([A|X],Z).
reduce(insert(A,X),Z):-reduce(X,[FX|RX]),reduce([FX|insert(A,RX)],Z).
reduce(append(X,Y),Z):-reduce(X,[],reduce(Y,Z).
reduce(append(X,Y),Z):-reduce(X,[U|V]),reduce([U|append(V,Y)],Z).
reduce(interleave(X,Y),Z):-reduce(X,[U|V]),reduce([U|interleave(Y,V)],Z).
reduce(a,Z):-reduce([1|a],Z).
reduce(b,Z):-reduce([2|b],Z).

```

If we now type, in Prolog, `reduce(perm([1,2,3]),Z)`, we obtain `Z=[1|perm([2,3])]`, `Z=[2|insert(1,perm([3]))]`, and `Z=[3|insert(1,insert(2,perm([])))]`. Note the following: First, `perm([1,2,3])` is only partially reduced, and *directly* by Prolog, not by some lazy interpreter implemented in Prolog. Second, the terms to which `Z` is bound are exactly those to which `perm([1,2,3])` is reduced by `select`. This illustrates Prolog simulating behavior of `select`. If we now define:

```

first(0,X,[]).
first(N,X,[FX|Z]):-not(N=0),reduce(X,[FX|RX]),N1 is N-1,first(N1,RX,Z).
make_list(E,[]):-reduce(E,[]).
make_list(E,[FE|Z]):-reduce(E,[FE|RE]),make_list(RE,Z).
print_list(X):-reduce(X,[FX|RX]),write(FX),write(' '),print_list(RX).

```

and then type, `make_list(perm([1,2,3]),Z)`, we obtain `Z=[1,2,3]`, `Z=[1,3,2]`, `Z=[2,1,3]`, `Z=[2,3,1]`, `Z=[3,1,2]`, `Z=[3,2,1]`. As further examples, if we type the queries on the left-hand side, we obtain the answers on the right-hand side:

```

reduce(append(a,b),Z) ---> Z=[1|append(a,b)]
reduce(interleave(a,b),Z) ---> Z=[1|interleave(b,a)]
first(5,interleave(a,b),Z) ---> Z=[1,2,1,2,1]
print_list(interleave(a,b)) ---> 1,2,1,2,1,2,.....

```

I.2.4 LOG(F)

LOG(F) is defined to be a logic programming system augmented with an F* compiler, and the equality axiom $X=X$. The result of compilation is to add to a logic programming system, a primitive for lazily simplifying F* terms. This primitive can be called from other Horn clauses, so LOG(F) is proposed as a combination of logic programming,

rewriting and lazy evaluation.

For problems for which lazy evaluation does not reduce lengths of computation, e.g. sorting, or all permutations, LOG(F) is empirically found to be about five times slower than Prolog. For problems for which lazy evaluation does reduce lengths of computation, e.g. N-queens, LOG(F) is faster than Prolog by unbounded, even infinite, amounts.

In the literature, [Vuillemin 1974, Berry & Levy 1979], optimality is used synonymously with minimality. Due to minimality of DF*, LOG(F) can also be said to be optimal. It can also be said to be so in a weaker sense, because of its desirable computational properties, and their economical realization in Prolog.

1.2.5 Applications of LOG(F)

LOG(F) can be used to do lazy functional programming in logic. In particular, it can be used to manipulate representations of infinite structures, such as in real analysis, exact real arithmetic, graphics, or networks of communicating processes.

The SKI rules of combinatory logic can be expressed as a DF* program. From confluence of DF*, a new proof is obtained of the confluence of combinatory logic.

DF* seems to offer a reasonable compromise between sequential execution and unbounded parallelism. Due to directedness of DF*, arguments of f in $f(t_1, \dots, t_m)$ can be simplified in parallel, however, they would be simplified lazily. Thus, DF* seems to be a good candidate for implementation on parallel machines.

Finally, if a DF* program is interpreted as an equality theory, reduce clauses can be thought of as implementing an equality theory in Prolog with the restriction that it be used only for simplification of terms. Now, given a clause of the form $p(c(X_1, \dots, X_m)):-\text{Body}$, where c is a constructor symbol, we can add another clause stating a rule of substitution of equals:

$$p(X):-\text{reduce}(X, c(X_1, \dots, X_m)), p(c(X_1, \dots, X_m)).$$

Now, even when a term E is not of the form $c(X_1, \dots, X_m)$, p can still be inferred for E , provided E is reducible to a term of the form $c(X_1, \dots, X_m)$. For example, with the Prolog rule for computing perimeters of regular polygons, $\text{peri}(\text{reg_poly}(N, S), Z):-Z \text{ is } N*S$, we can infer $\text{peri}(\text{reg_poly}(3, 10), 30)$. We can now add the clause:

$$\text{peri}(X, Y):-\text{reduce}(X, Z), \text{peri}(Z, Y).$$

Where `reg_poly` is a constructor, and `equi`, `square`, and `hexagon` are non-constructors, an equality theory among polygons, expressed in F*, is:

```
equi(S)=>reg_poly(3,S).
square(S)=>reg_poly(4,S).
hexagon(S)=>reg_poly(6,S).
```

This is compiled into:

```
reduce(reg_poly(A,B),reg_poly(A,B)).
reduce(equi(S),Z):-reduce(reg_poly(3,S),Z).
reduce(square(S),Z):-reduce(reg_poly(4,S),Z).
reduce(hexagon(S),Z):-reduce(reg_poly(6,S),Z).
```

The Prolog query `peri(equi(10),30)`, now succeeds. Thus Prolog automatically infers the result of substituting `equi(10)` for `reg_poly(3,10)`, in `peri(reg_poly(3,10),30)`. Of course, if we type `peri(square(3),Z)`, we obtain `Z=12`.

1.3 RELATIONSHIP WITH PREVIOUS WORK

There seem to be two major approaches to combining logic programming, and rewriting. The first consists of implementing logic programming in rewriting, e.g. `LOGLISP` [Robinson & Sibert 1982], or `QLOG` [Komorowski 1982]. However, it seems difficult for such an approach to lead to an efficient system since logic programs must pass through two high-level layers of interpretation.

The second approach consists of developing a new computational model of which both rewriting, and logic programming are instances. Examples of such models include those based upon semantic- or T-unification, [Goguen & Meseguer 1986], [Subrahmanyam & You 1984], [Kornfeld 1983], sets, [Robinson 1987], [Darlington et al. 1986], narrowing, [Reddy 1985], the Knuth-Bendix completion procedure, [Dershowitz & Josephson 1984], oriented equational clauses, [Fribourg 1984], residuation, [Ait-Kaci & Nasr 1987], extension of SLD-resolution with narrowing, [Yamamoto 1987], or extension of SLD-resolution with atom-elimination rule, [Barbuti et al. 1986].

In order for a new computational model to be satisfactory, it must possess not only good declarative semantics, but also good procedural semantics. The former is essential for reasoning about programs in the model. The latter means that the behavior of the model is simple enough that it can be visualized, predicted, and controlled. It is essential if the model is to be used for programming, i.e. for expressing algorithms. In this regard, we also quote Robinson [1984]:

...one guiding principle must surely be that logic programming, however narrowly or broadly construed, essentially involves the ingredient of practicality. The underlying deductive processes should have enough directness and predictability to permit the planning of efficient logical computations. Herein probably lies the important distinction, difficult to make precise but nonetheless real, between logic programming proper and automatic deduction in general.

However, developing a satisfactory computational model, more general than logic programming and rewriting, is a very ambitious undertaking, particularly if the model is also to exhibit laziness. In particular, it appears that each of the above models, with the possible exception of [Robinson 1987], and [Darlington et al. 1986], either has complex declarative semantics, or complex procedural semantics.

Of course, even if a satisfactory computational model is developed, its efficient implementation on concrete machines can still pose a considerable software engineering challenge, requiring several person-years of effort. In particular, it appears that efficient implementation of the above proposals is still an ongoing effort.

Lazy evaluation itself does not seem to be easy to implement efficiently. Several implementations of lazy evaluation for functional, and logic-based languages have been proposed e.g. [Friedman & Wise 1976, Henderson 1980, Turner 1979, O'Donnell 1985, Clark & McCabe 1979, Hansson et al. 1982, Shapiro 1983, Barbuti et al. 1986]. However, only a few of these systems, e.g. Turner's, or O'Donnell's, seem to be efficient enough for practical programming.

In view of such difficulties with developing, and implementing a new computational model of which logic programming, rewriting, and lazy evaluation, are instances, we ask whether it is possible to *subsume* the last two within the first. In other words, we ask whether it is possible to keep SLD-resolution fixed, but use it in such a way that it performs, *in a computationally feasible manner*, rewriting, and lazy evaluation? If such an attempt were to succeed, we would not only obtain a declarative semantics of rewriting, and lazy evaluation using purely logical ideas, we would also have a very efficient implementation of these, in, say, Prolog.

Important precedents in this direction have already been established with the subsumption, within logic programming, of grammars, and relational databases. Definite clause grammar rules [Pereira & Warren 1980] can be expressed as Horn clauses in such a way that their interpretation using Prolog, directly simulates top-down parsing. Relational databases can be expressed directly as ground Horn clauses [Gallaire & Minker 1978]. Prolog enables inference with them in ways (e.g. using recursion) not possible with conventional data retrieval operators.

An important step towards subsuming rewriting within logic programming, has recently been taken by van Emden & Yukawa [1987], whose motivations are very similar to, but independent, of ours. They show how to derive logical consequences of the standard equality axioms which result in a small SLD-search space. They also show how to compile an equality theory into equality free Horn clauses, which also result in a small SLD-search space. *However, their approach is restricted only to terminating equality theories.* These are insufficient for representing infinite structures.

As pointed out in [Narain 1986], the compactness theorem of first order logic [Robinson 1979] suggests that lazy evaluation is already present in first order logic. It states that if an infinite set of clauses is unsatisfiable then it has a finite subset which is also unsatisfiable. Moreover, a complete proof procedure, such as SLD-resolution for Horn clauses would find this set in finite time. Thus, as with lazy evaluation, one could get termination in finite time even with an infinite input.

This idea was investigated further, and led to a method in [Narain 1986], for defining functions by Horn clauses in such a way that when SLD-resolution interprets these, it behaves lazily. However, the discussion is limited mainly to lists, although a generalization to other data structures is hinted.

The current system, LOG(F), is an attempt to generalize, and develop a purely syntactic explanation of the above method. It appears to subsume within logic programming, in a rigorous yet computationally efficient fashion, non-terminating, non-deterministic rewriting, and lazy evaluation.

Minimality of DF* appears to be a generalization of similar results by Vuillemin [1974], and Berry & Levy [1979]. Both derive it only for rewrite rules whose left hand sides are of the form $f(X_1, \dots, X_m)$, where each X_i is a variable. Thus, they must assume existence of a finite number of primitive functions such as if-then-else, which are not definable using such rules alone. In contrast, F* admits rewrite rules in which the X_i can be patterns. Thus, in F*, as in logic programming, it is not necessary to assume existence of any primitive functions.

Restrictions on rewrite rules in F*, and the reduction strategy select, seem to be substantially simpler than their counterparts in the system of O'Donnell [1985]. Select also seems to be substantially simpler than its counterpart in the system of Huet & Levy [1979]. Furthermore, since F* can be compiled into efficient Horn clauses, and Prolog can be used, implementation of F* is straightforward. However, implementation of the other two systems seems to be quite a major undertaking.

Confluence of DF* is anticipated by Huet [1980] who derives sufficient conditions for

confluence for rewrite rule systems more general than DF^* . However, our proof, being specialized for DF^* , is very simple.

$LOG(F)$ bears only a superficial similarity to the system of Tamaki [1984]. He also shows how to compile equality theories into Horn clauses with a smaller search space. However, as is pointed out in his paper, these clauses can still be seriously inefficient, particularly, when manipulating representations of infinite structures. As shown in Section VI.8, such inefficiency is not exhibited by $LOG(F)$. Also, his reducibility predicate can terminate even without simplifying terms whereas that of $LOG(F)$ cannot.

I.4 OUTLINE OF PAPER

Section II defines F^* , and the reduction strategy $select$, and establishes its reduction-completeness. Section III defines DF^* , and shows its confluence and directedness. Section IV defines Labeled DF^* , a subset of DF^* , for the purpose of formalizing the notion of a copy of a term, and then establishes its minimality. Section V describes an algorithm for compiling F^* into Horn clauses, and proves its correctness. Section VI describes examples of programming in $LOG(F)$, and compares performance of $LOG(F)$ with that of Prolog. Section VII contains a summary and conclusions. Proofs of relatively minor propositions have been omitted, or abbreviated. These can be obtained in full in [Narain 1988].

II. A REWRITE RULE SYSTEM F^*

II.1 INTRODUCTION

A first order, non-deterministic, non-terminating rewrite rule system F^* , and a lazy reduction strategy for it, *select*, are defined. The emphasis in F^* is on computing simplified forms, instead of normal forms. Thus, termination problems faced by certain previous approaches are avoided.

The main result proved is that F^* is reduction-complete, in that *select* reduces ground terms to their simplified, or normal forms, whenever possible. Reduction-completeness yields a weak form of laziness. A term may denote an infinite object, and so fail to have a finite normal form. However, if it has a finite simplified form, it is obtained in finite time. By repeatedly simplifying subterms of this simplified form, the structure of the infinite object can be revealed to any arbitrary depth.

II.2 DEFINITION OF F^*

Variables. There is a countably infinite list of variables.

Function symbols. There is a countably infinite list of 0-ary function symbols. In particular, `[]`, `0`, `true`, `false`, are 0-ary function symbols. There is a countably infinite list of 1-ary function symbols. In particular, `s` is a 1-ary function symbol. There is a countably infinite list of 2-ary function symbols. In particular, `!` is a 2-ary function symbol. And so on, for all other arities.

Connectives. The connectives are `=>`, `(,)`, `'`, `,`.

Constructor Symbols. There is an infinite subset of the function symbols called Constructors. Each element of Constructors is called a constructor symbol. For each n , $n \geq 0$, Constructors contains an infinite number of n -ary function symbols. In particular, `0`, `true`, `false`, `[]` and `!` are constructor symbols. It is intended that data be represented by combinations of only constructor symbols.

Terms. A term is either a variable, or an expression of the form $f(t_1, \dots, t_n)$ where f is an n -ary function symbol, $n \geq 0$, and each t_i is a term. A term is called ground if it contains no variables. *It is the intention in F^* to reduce only ground terms*, and most of the propositions below are about these. Non-ground terms such as left hand sides of reduction rules do arise, but in very few propositions. **Hence, unless explicitly stated otherwise, by a term is meant a ground term.**

Subterms. Let E be a term. Then E is said to be a subterm of itself. Also, if $E=f(t_1, \dots, t_n)$, $n > 0$, then X is said to be a subterm of E , if X is a subterm of some t_i . Let X be a subterm of E . Then X is said to occur in E . Also, if $X \neq E$, then X is said to be a proper subterm of E , or be properly contained in E . Two subterms A and B of E are said to overlap, if A is properly contained within B .

Substitutions. A substitution is a, possibly empty, set $\{ \langle X_1, t_1 \rangle, \dots, \langle X_n, t_n \rangle \}$ where the X_1, \dots, X_n are distinct variables, and each t_i is a term, possibly containing variables. A variable X is defined in a substitution σ iff for some possibly non-ground term s , $\langle X, s \rangle$ occurs in σ . In this paper, we will be concerned almost exclusively with substitutions in which for each pair $\langle X, s \rangle$, s is a ground term.

Applying substitutions to terms. Let $\sigma = \{ \langle X_1, t_1 \rangle, \dots, \langle X_n, t_n \rangle \}$ be a substitution and E be a term, possibly containing variables. The result of applying σ to E , $E\sigma$, is the result of replacing, for each i , every occurrence of X_i in E by t_i .

Matching. A ground term E is said to **match** a possibly non-ground term F , with substitution α , if $E = F\alpha$.

Unification. Two terms, E and F , possibly containing variables, are said to unify with substitution σ if $E\sigma = F\sigma$. Note that matching is a special case of unification.

Reduction Rules. A reduction rule is of the form:

$$\text{LHS} \Rightarrow \text{RHS}$$

where LHS and RHS are terms, possibly containing variables. LHS is called the head of the rule. The following restrictions are placed on LHS and RHS:

- (a) LHS is not a variable.
- (b) LHS is not of the form $c(t_1, \dots, t_n)$ where c is a constructor symbol.
- (c) If $\text{LHS} = f(t_1, t_2, \dots, t_n)$, then each t_i is a variable, or a term of the form $c(X_1, \dots, X_m)$ where c is an m -ary constructor symbol, and each X_i a variable.
- (d) There is at most one occurrence of any variable in LHS.
- (e) All variables of RHS appear in LHS.

These restrictions are very reasonable, and as examples throughout the paper show, very

expressive programs can be written adhering to these. *Note that F* is more expressive than first order Lisp, as the latter does not admit patterns in left hand sides of function definitions.*

Restriction (a) is to enable functional programs to be written in F*.

Restriction (b) ensures that a term of the form $c(t_1, \dots, t_n)$, c a constructor symbol, cannot be reduced as a whole. This yields a simple halting condition for the basic simplification process. If further simplification is required, the process may be called recursively.

Restriction (c) limits heads of rules to be of depth at most two, and so greatly simplifies analysis. However, no generality is lost, since rules with heads of arbitrary depth can easily be expressed in terms of rules with heads of depth at most two. For example, the rule:

$$\text{fib}(s(s(X))) \Rightarrow \text{plus}(\text{fib}(X), \text{fib}(s(X)))$$

can be expressed as:

$$\begin{aligned} \text{fib}(s(A)) &\Rightarrow g(A) \\ g(s(X)) &\Rightarrow \text{plus}(\text{fib}(X), \text{fib}(s(X))). \end{aligned}$$

Restriction (d) is the linearity assumption. It ensures that to match a ground term $f(t_1, \dots, t_n)$ with the left hand side of a rule $f(L_1, \dots, L_n)$, it is sufficient to match, for each i , t_i with L_i .

Restriction (e) ensures that a ground term is never reduced to a non-ground term. Again, this is necessary if F* is to be used for functional programming.

F* programs. An F* program is a finite set of reduction rules. Where t is a binary constructor symbol, some examples of F* programs are:

$$\begin{aligned} \text{quicksort}([]) &\Rightarrow [] \\ \text{quicksort}([A|B]) &\Rightarrow \text{quicksort1}(A, \text{partition}(A, B, [], [])) \\ \text{quicksort1}(A, t(L, R)) &\Rightarrow \text{append}(\text{quicksort}(L), [A|\text{quicksort}(R)]) \\ \\ \text{partition}(U, [], L, R) &\Rightarrow t(L, R) \\ \text{partition}(U, [A|B], L, R) &\Rightarrow \\ &\quad \text{if}(\text{lesseq}(A, U), \text{partition}(U, B, [A|L], R), \text{partition}(U, B, L, [A|R])) \\ \\ \text{append}([], X) &\Rightarrow X \end{aligned}$$

$\text{append}([U|V],W) \Rightarrow [U|\text{append}(V,W)]$

$\text{if}(\text{true},X,Y) \Rightarrow X.$

$\text{if}(\text{false},X,Y) \Rightarrow Y.$

$\text{lesseq}(0,X) \Rightarrow \text{true}.$

$\text{lesseq}(s(X),s(Y)) \Rightarrow \text{lesseq}(X,Y).$

$\text{lesseq}(s(X),0) \Rightarrow \text{false}.$

$\text{zero}(X) \Rightarrow 0.$

$\text{prim_rec_f}(0,Y1,Y2,Y3) \Rightarrow g(Y1,Y2,Y3).$

$\text{prim_rec_f}(s(X),Y1,Y2,Y3) \Rightarrow h(\text{prim_rec_f}(X,Y1,Y2,Y3),X,Y1,Y2,Y3).$

$\text{minim_p}(X,K) \Rightarrow \text{if}(\text{equal}(p(X),K),X,\text{minim_p}(s(X),K)).$

$\text{equal}(0,0) \Rightarrow \text{true}.$

$\text{equal}(0,s(X)) \Rightarrow \text{false}.$

$\text{equal}(s(X),0) \Rightarrow \text{false}.$

$\text{equal}(s(X),s(Y)) \Rightarrow \text{equal}(X,Y).$

$\text{merge}([A|B],[C|D]) \Rightarrow \text{if}(\text{lesseq}(A,C),[A|\text{merge}(B,[C|D])],[C|\text{merge}([A|B],D)]).$

$\text{int}(N) \Rightarrow [N|\text{int}(s(N))].$

$\text{greater}(X,Y) \Rightarrow \text{not}(\text{lesseq}(X,Y)).$

$\text{not}(\text{true}) \Rightarrow \text{false}.$

$\text{not}(\text{false}) \Rightarrow \text{true}.$

We now consider the reduction of terms. Again, unless explicitly stated, by a term we mean a ground term.

$E \Rightarrow_P E1$. Let P be an F^* program and E and $E1$ be terms. We say $E \Rightarrow_P E1$ if there is a rule $\text{LHS} \Rightarrow \text{RHS}$ in P , and a substitution σ such that $E = \text{LHS}\sigma$, and $E1 = \text{RHS}\sigma$. We also say that E reduces to $E1$ by the rule $\text{LHS} \Rightarrow \text{RHS}$, or that the rule applies to the whole of E . The subscript on \Rightarrow is dropped, if clear from context.

$F = E[G/H]$. Where E, F, G, H , are terms, let F be the result of replacing an occurrence of G in E by H . Then we say $F = E[G/H]$.

$E \rightarrow_P E1, E \rightarrow_P^* E1$. Let P be an F^* program and E be a term. Let G be a subterm of E such that $G \Rightarrow_P H$. Let $E1$ be the result of substituting H for G in E . Then we say that E

$\rightarrow_P E1$. Note that if $E \Rightarrow_P E1$ then E matches the left hand side of some rule in P . If $E \rightarrow_P E1$ then some subterm of E , including possibly E , matches the left hand side of some rule in P . We define \rightarrow_P^* to be the reflexive transitive closure of \rightarrow_P . Again, the subscript on \rightarrow or \rightarrow^* is dropped, if clear from context.

Reductions. Let P be an F^* program. A reduction in P is a, possibly infinite, sequence $E1, E2, \dots$ such that for each i , when E_i and E_{i+1} both exist, $E_i \rightarrow_P E_{i+1}$.

Lengths of reductions. The length of a finite reduction $E0, E1, \dots, E_n$ is n .

Simplified forms. A term is said to be in simplified form or simplified if it is of the form $c(t1, \dots, tn)$ where c is an n -ary constructor symbol, $n \geq 0$, and each t_i is a term. F is called a simplified form of E , if $E \rightarrow_P^* F$ and F is in simplified form.

Normal forms. A term is said to be in normal form if each function symbol in it is a constructor symbol. F is called a normal form of E if $E \rightarrow_P^* F$ and F is in normal form.

Successful reductions. Let P be an F^* program. A successful reduction in P is a finite reduction $E0, \dots, E_n$, $n \geq 0$, in P , such that E_n is simplified.

$R_P(G, H, A, B)$. Let P be an F^* program. Where G, H, A, B are terms, $R_P(G, H, A, B)$ if (a) $G \Rightarrow H$, and (b) B is identical with A except that zero or more occurrences of G in A are *simultaneously* replaced by H . Note that A and G can be identical. Again, if P is clear from context we omit the subscript on R .

Reduction strategy. Let P be an F^* program. A reduction strategy for P takes as input a term E and selects a subterm G of E such that there exists a term H such that $G \Rightarrow_P H$.

A special reduction strategy. Let P be an F^* program. We now define a reduction strategy, select_P for P . Informally, given a term E it will select that subterm of E whose reduction is necessary in order that some \Rightarrow rule in P apply to the whole of E . Where $f(T1, \dots, Tn)$ is a term, the relation select_P is defined by the following pseudo-Horn clauses:

$$\begin{aligned} &\text{select}_P(f(T1, \dots, Tn), f(T1, \dots, Tn)) \text{ if } f(T1, \dots, Tn) \Rightarrow_P X. \\ &\text{select}_P(f(T1, \dots, Ti, \dots, Tn), X) \text{ if} \\ &\quad \text{there is a rule } f(L1, \dots, Li, \dots, Ln) \Rightarrow \text{RHS} \text{ in } P, \text{ and} \\ &\quad \text{there is no substitution } \sigma \text{ such that } Ti = Li\sigma, \text{ and} \\ &\quad \text{select}_P(Ti, X). \end{aligned}$$

The second rule is a schema, so that an instance of it is assumed written for each i , $1 \leq i \leq n$. Again, the subscript on select is dropped, if clear from context. Note the

following:

- (1) When select_P takes as input E and returns G , it also, implicitly, computes a position, or occurrence of G in E . This occurrence can be obtained from the proof of $\text{select}_P(E, G)$.
- (2) If $\text{select}_P(E, G)$, there is a term H such that $G \Rightarrow_P H$.
- (3) Select is non-deterministic, in that given term E , it is possible for it to select more than one subterm A_1, \dots, A_k , $k > 0$, within E . Also, it is possible that for some i, j , $i \neq j$, A_i is a proper subterm of A_j .
- (4) Since, by restriction (b) there is no rule in P of the form $c(t_1, \dots, t_n) \Rightarrow \text{RHS}$, where c is a constructor symbol, if E is simplified, select_P is undefined for E .

For example, where P is the set of reduction rules which appear above, and 1, 2, ... are abbreviations, respectively, for $s(0), s(s(0)), \dots$, we have the following:

```

select(merge(int(1),int(2)),int(1)).
select(merge(int(1),int(2)),int(2)).
select(merge([1,3],int(2)),int(2)).
select(merge([1,2],[3,4]),merge([1,2],[3,4])).
If  $E = [1 \mid \text{merge}(\text{int}(1), \text{int}(2))]$  then select is undefined for  $E$ .
select(lesseq(0,zero(1)),lesseq(0,zero(1))).
select(lesseq(0,zero(1)),zero(1)).

```

Let E, G, H be terms. In the following, when we say that $\text{select}(E, G)$, and G is to be replaced by H in E , we mean that the occurrence of G derived from the proof of $\text{select}(E, G)$, is to be replaced by H .

N-step. Let P be an F^* program and E, G, H be terms. Let $\text{select}_P(E, G)$, and $G \Rightarrow_P H$. Let E_1 be the result of replacing G by H in E . Then we say that E reduces to E_1 in an N -step in P . The qualification "in P " is omitted when P is clear from context. The prefix N in N -step is intended to connote normal order.

N-reduction. Let P be an F^* program. An N -reduction in P is a reduction E_1, E_2, \dots in P such that for each i , when E_i and E_{i+1} both exist, E_i reduces to E_{i+1} in an N -step in P . In particular, the sequence E where E is a term, is an N -reduction in P . The qualification "in P " is omitted when P is clear from the context.

select-r. This reduction strategy repeatedly uses select to reduce terms. The suffix r

stands for recursive, or repeated. Where P is an F^* program:

$$\begin{aligned} &\text{select-}r_P(E,F) \text{ if } \text{select}_P(E,F). \\ &\text{select-}r_P(c(T_1,\dots,T_i,\dots,T_m),F) \text{ if} \\ &\quad c \text{ is a constructor symbol, and} \\ &\quad \text{select-}r_P(T_i,F). \end{aligned}$$

Again, the second rule is a schema, so that an instance of it is assumed written for each i , $1 \leq i \leq n$. Thus, $\text{select-}r$ is like select except that if a term is in simplified form, it recursively uses select on one of the arguments of the outermost constructor symbol. So, its repeated use can yield normal-forms of terms.

For example, with the usual rules for append , the query $\text{select}([\text{1append}([],[])],X)$ fails, whereas the query $\text{select-}r([\text{1append}([],[])],X)$ succeeds with $X=\text{append}([],[])$. The subscript on $\text{select-}r$ is dropped, if clear from context.

NR-step. Let P be an F^* program and E,G,H be terms. Suppose $\text{select-}r_P(E,G)$ and $G \Rightarrow_P H$. Let E_1 be the result of replacing G by H in E . Then we say that E reduces to E_1 in an NR-step in P . The qualification "in P " is omitted when clear from context.

NR-reduction. Let P be an F^* program. An NR-reduction in P is a reduction E_1, E_2, \dots in P such that for each i , when E_i and E_{i+1} both exist, E_i reduces to E_{i+1} in an NR-step in P . In particular, the sequence E where E is a term, is an NR-reduction in P .

NR-reductions are needed to compute normal-forms of terms. For example, the term $\text{append}([1],[2])$ has the only N-reduction $\text{append}([1],[2]), [\text{1append}([],[2])]$. However, it has the NR-reduction $\text{append}([1],[2]), [\text{1append}([],[2])], [1,2]$. The qualification "in P " is omitted when clear from context.

II.3 REDUCTION-COMPLETENESS OF F^*

Lemma 1. Let P be an F^* program. If $A \rightarrow B$ and B is simplified but A is not, then $A \Rightarrow B$.

Proof. Clear, from restriction (b). QED.

Lemma 2. Let P be an F^* program. Let X_1, \dots, X_n be variables, $G, H, t_1, \dots, t_n, t_1^*, \dots, t_n^*$ be terms such that for each i , $R(G, H, t_i, t_i^*)$. Let $\sigma = \{ \langle X_1, t_1 \rangle, \dots, \langle X_n, t_n \rangle \}$ and $\tau = \{ \langle X_1, t_1^* \rangle, \dots, \langle X_n, t_n^* \rangle \}$ be substitutions. Let M be a term, possibly containing variables, but only from $\{X_1, \dots, X_n\}$. Then $R(G, H, M\sigma, M\tau)$.

Proof. By induction on length of M . QED.

Lemma 3. Let P be an F^* program. If:

- (1) $G, H, E1=f(t1,...,tn)$ and $F1=f(t1^*,...,tn^*)$ are terms, and
- (2) $R(G,H,ti,ti^*)$ for every i in $1,...,n$, and
- (3) $B=f(L1,...,Ln)$ is the head of some rule in P , and
- (4) $E1=B\sigma$ for some substitution σ , which defines only the variables in B .

Then there exists a substitution τ such that:

- (1) $F1=B\tau$, and
- (2) σ and τ define exactly the same variables, and
- (3) If pair $\langle X,s \rangle$ occurs in σ , and $\langle X,s^* \rangle$ occurs in τ , then $R(G,H,s,s^*)$.

Proof. Clear, by restrictions (a)-(e). QED.

Lemma 4. Let P be an F^* program. If:

- (1) $f(t1,...,ti,...,tn)$ is a term, and
- (2) $f(L1,...,Li-1,c(X1,...,Xm),Li+1,...,Ln) \Rightarrow RHS$ is a rule in P , and
- (3) $ti=d1,d2,d3,...,dr, r>0$, is an N-reduction.

Then, $f(t1,...,ti-1,d1,ti+1,...,tn), f(t1,...,ti-1,d2,ti+1,...,tn), \dots, f(t1,...,ti-1,dr,ti+1,...,tn)$ is also an N-reduction.

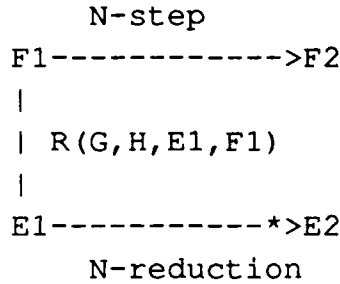
Proof. By definition of N-reduction. QED.

Theorem 1. Let P be an F^* program. Let $E1,F1,F2,G,H$ be terms such that

- (1) $R(G,H,E1,F1)$, and
- (2) $F1$ reduces to $F2$ in an N-step

Then there is an N-reduction $E1,...,E2$ in P such that $R(G,H,E2,F2)$.

Proof. It is helpful to draw the following diagram:



We have to show that $R(G, H, E2, F2)$. We proceed by induction on length of $E1$. Suppose $E1$ is a 0-ary function symbol. If $E1 = F1$ then $E1, F2$ is an N-reduction and $R(G, H, F2, F2)$. If $E1 \neq F1$ then since $R(G, H, E1, F1)$, $E1 = G$ and $E1 \Rightarrow F1$. Thus, there is an N-reduction $E1, F1, F2$ and $R(G, H, F2, F2)$. In both cases, take $E2 = F2$.

Otherwise, $E1 = f(t1, \dots, tn)$, $n > 0$. Assume the theorem for every term whose length is less than that of $f(t1, \dots, tn)$. If $E1 = F1$ then $E1, F2$ is an N-reduction and $R(G, H, F2, F2)$. Otherwise $E1 \neq F1$. If $E1 = G$ then since $R(G, H, E1, F1)$, $E1 \Rightarrow F1$. Thus, there is an N-reduction $E1, F1, F2$, and $R(G, H, F2, F2)$. Again, in both cases, take $E2 = F2$.

We now arrive at the interesting cases, with $E1 \neq F1$, but $G \neq E1$. Hence $F1 = f(t1^*, \dots, tn^*)$ where for every i , $R(G, H, ti, ti^*)$. We now consider the following cases:

Case 1. $F1 \Rightarrow F2$. Then there is a rule $f(L1, \dots, Ln) \Rightarrow \text{RHS}$ in P , such that $F1$ matches $f(L1, \dots, Ln)$ with substitution τ , and $F2 = \text{RHS}\tau$.

Case 1-1. $E1$ matches $f(L1, \dots, Ln)$ with substitution σ . By Lemma 3, there exists substitution β such that $F1 = f(L1, \dots, Ln)\beta$. Since $F1 = f(L1, \dots, Ln)\tau$, $\tau = \beta$.

$E1 \Rightarrow \text{RHS}\sigma$, so let $E2 = \text{RHS}\sigma$. The N-reduction is $E1, E2$. Of course $F2 = \text{RHS}\tau$. By Lemma 3, σ and τ define exactly the same variables, and if $\langle X, s \rangle$ occurs in σ and $\langle X, s^* \rangle$ appears in τ then $R(G, H, s, s^*)$. Hence, by Lemma 2, $R(G, H, E2, F2)$.

Case 1-2. $E1$ does not match $f(L1, \dots, Ln)$. Then, since $E1$ is ground and each variable occurs at most once in $f(L1, \dots, Ln)$, there is some Li in $L1, \dots, Ln$, and some ti in $t1, \dots, tn$, such that ti does not match Li . Hence Li is not a variable, so $Li = c(X1, \dots, Xm)$, c a constructor symbol and each Xi a variable.

Moreover, since $R(G, H, ti, ti^*)$, and ti does not match Li , by restriction (c), ti is not simplified. Since $F1$ matches $f(L1, \dots, Ln)$, ti^* matches Li , and so ti^* is simplified. Since $R(G, H, ti, ti^*)$, $ti \Rightarrow ti^*$. Thus $\text{select}(E1, ti)$. Hence $f(t1, \dots, ti, \dots, tn)$ reduces to

$f(t_1, \dots, t_i^*, \dots, t_n)$ in an N-step.

Hence there exists an N-reduction $E_1 = P_1, P_2, P_3, \dots$ such that for each i , $P_i = f(s_1, \dots, s_n)$, and for each s_k in s_1, \dots, s_n , $s_k = t_k$ or $s_k = t_k^*$. Moreover, P_{i+1} is derived from P_i by selecting some s_k in s_1, \dots, s_n such that s_k does not match L_k in L_1, \dots, L_n , and replacing s_k , in P_i , by t_k^* . We also have for each i , $R(G, H, P_i, F_1)$. Since n is finite, this reduction cannot be infinite and must end in P_m such that P_m matches $f(L_1, \dots, L_n)$ with substitution σ . Then $P_m \Rightarrow \text{RHS}\sigma$. Hence we have the N-reduction $E_1, P_2, P_3, \dots, P_m, \text{RHS}\sigma$. Take $E_2 = \text{RHS}\sigma$. By Lemma 3, F_1 and $f(L_1, \dots, L_n)$ match with some substitution, and clearly this is τ . Already, $F_2 = \text{RHS}\tau$. By Lemma 2, $R(G, H, E_2, F_2)$.

Case 2. Not $F_1 \Rightarrow F_2$. We are given that F_1 reduces to F_2 by an N-step. We now have to show that there is an N-reduction E_1, \dots, E_2 such that $R(G, H, E_2, F_2)$.

Suppose $\text{select}(F_1, u)$. Then u occurs in some t_i^* . That is, there is some t_i^* in t_1^*, \dots, t_n^* , such that $\text{select}(t_i^*, u)$. Let $u \Rightarrow v$ and let t_i^{**} be the result of replacing u in t_i^* by v . Hence t_i^* reduces to t_i^{**} in an N-step, and also $F_2 = f(t_1^*, \dots, t_i^{**}, \dots, t_n^*)$. By definition of select , there is a rule $f(L_1, \dots, L_i, \dots, L_n) \Rightarrow \text{RHS}$ in P such that t_i^* does not match L_i . Hence $L_i = c(X_1, \dots, X_m)$, $m \geq 0$, where c is a constructor symbol and each X_i is a variable.

Clearly, t_i^* is not simplified. So, by restriction (b) t_i is also not simplified. t_i^* reduces to t_i^{**} in an N-step. We already have $R(G, H, t_i, t_i^*)$. Since the length of t_i is less than that of $f(t_1, \dots, t_i, \dots, t_n)$, by induction hypothesis there is an N-reduction $t_i = d_1, d_2, \dots, d_r, r \geq 1$, such that $R(G, H, d_r, t_i^{**})$. By Lemma 4, the sequence $f(t_1, \dots, t_{i-1}, t_i, t_{i+1}, \dots, t_n)$, $f(t_1, \dots, t_{i-1}, d_2, t_{i+1}, \dots, t_n), \dots, f(t_1, \dots, t_{i-1}, d_r, t_{i+1}, \dots, t_n)$ is an N-reduction. Take $E_2 = f(t_1, \dots, t_{i-1}, d_r, t_{i+1}, \dots, t_n)$. We already have $F_2 = f(t_1^*, \dots, t_i^{**}, \dots, t_n^*)$ and for each k , $R(G, H, t_k, t_k^*)$. Hence $R(G, H, E_2, F_2)$. QED.

Lemma 5. Let P be an F^* program. Let $R(G, H, E_0, F_0)$ and F_0, F_1, \dots, F_n be an N-reduction. Then there is an N-reduction $E_0, \dots, E_1, \dots, E_n$ such that $R(G, H, E_n, F_n)$.

Proof. By induction on length n of F_0, F_1, \dots, F_n . If $n=0$ then clear. Otherwise assume lemma for the N-reduction F_1, \dots, F_n . Since F_0 reduces to F_1 in an N-step and $R(G, H, E_0, F_0)$, by Theorem 1, there exists an N-reduction E_0, \dots, E_1 such that $R(G, H, E_1, F_1)$. By induction hypothesis, there exists an N-reduction E_1, \dots, E_n , such that $R(G, H, E_n, F_n)$. Hence there exists the N-reduction $E_0, \dots, E_1, \dots, E_n$ such that $R(G, H, E_n, F_n)$. QED.

Theorem 2. Reduction-completeness of F^* for simplified forms. Let P be an F^* program and D_0 a term. Let $D_0, D_1, \dots, D_n, n \geq 0$, be a successful reduction in P . Then

there is a successful N-reduction D_0, E_1, \dots, E_m in P , such that $E_m \rightarrow^* D_n$.

Proof. By induction on length n of D_0, D_1, \dots, D_n . If $n=0$, D_0 is already simplified, so D_0 is a successful N-reduction, and $D_0 \rightarrow^* D_0$.

Let $n>0$ and assume Theorem for D_1, \dots, D_n . Then there is a successful N-reduction D_1, F_2, \dots, F_p such that $F_p \rightarrow^* D_n$. The situation can be laid out as follows:

$$\begin{array}{c}
 D_n \\
 : \\
 : \\
 D_2 \\
 | \\
 D_1 \rightarrow F_2 \rightarrow^* F_p \\
 | \\
 | \quad R(G, H, E_m, F_p), \quad F_p \rightarrow^* D_n \\
 | \\
 D_0 \rightarrow E_1 \rightarrow^* E_m
 \end{array}$$

Since $D_0 \rightarrow D_1$, there are terms G, H , such that $G \Rightarrow H$ and $D_1 = D_0[G/H]$. Hence $R(G, H, D_0, D_1)$. Since D_1, F_2, \dots, F_p is a successful N-reduction, by Lemma 5, there is an N-reduction D_0, E_1, \dots, E_q such that $R(G, H, E_q, F_p)$. If E_q is simplified, take $E_m = E_q$. Now D_0, E_1, \dots, E_m is a successful N-reduction. Since $R(G, H, E_m, F_p)$, and $F_p \rightarrow^* D_n$, $E_m \rightarrow^* D_n$, as required.

If E_q is not simplified, then since $R(G, H, E_q, F_p)$, and F_p is simplified, $E_q \Rightarrow F_p$. Now take $E_m = F_p$, so $D_0, E_1, \dots, E_q, E_m$ is a successful N-reduction, and $E_m \rightarrow^* D_n$, as required.
QED.

Theorem 3. Let P be an F^* program. Let E_1, F_1, F_2, G, H be terms such that

- (1) $R(G, H, E_1, F_1)$, and
- (2) F_1 reduces to F_2 in an NR-step

Then there is an NR-reduction E_1, \dots, E_2 in P such that $R(G, H, E_2, F_2)$.

Proof. By induction on length of E_1 . Let E_1 be a 0-ary function symbol. If $E_1 = F_1$ then clear. If $E_1 \neq F_1$, then $E_1 = G$, and so, clear. Otherwise, let $E_1 = f(t_1, \dots, t_n)$, $n>0$. Assume theorem for t_1, \dots, t_n .

Case 1. E_1 is unsimplified. If F_1 is simplified, then since $R(G, H, E_1, F_1)$, $E_1 = G$, so the

theorem is clear. If F_1 is unsimplified, then by definition of NR-reduction, F_1 reduces to F_2 in an N-step. By Theorem 1, there exists an N-reduction E_1, \dots, E_2 such that $R(G, H, E_2, F_2)$. But this is also an NR-reduction.

Case 2. E_1 is simplified. Then, since $R(G, H, E_1, F_1)$, F_1 is also simplified. Let $F_1 = f(s_1, \dots, s_n)$ where f is a constructor symbol. Hence for each i , $1 \leq i \leq n$, $R(G, H, t_i, s_i)$. Since F_1 reduces to F_2 in an NR-step, there is some s_i in s_1, \dots, s_n , such that s_i reduces to some s_i^* in an NR-step and $F_2 = f(s_1, \dots, s_i^*, \dots, s_n)$. By induction hypothesis, there exists an NR-reduction $t_i = t_{i1}, t_{i2}, \dots, t_{ik}$ such that $R(G, H, t_{ik}, s_i^*)$. It can easily be shown that the reduction $f(t_1, \dots, t_{i1}, \dots, t_n), f(t_1, \dots, t_{i2}, \dots, t_n), \dots, f(t_1, \dots, t_{ik}, \dots, t_n)$ is also an NR-reduction. Clearly $R(G, H, f(t_1, \dots, t_{ik}, \dots, t_n), F_2)$. QED.

Lemma 6. Let P be an F^* program. Let $R(G, H, E_0, F_0)$ and F_0, \dots, F_n , $n \geq 0$, be an NR-reduction. Then there is an NR-reduction E_0, \dots, E_k such that $R(G, H, E_k, F_n)$.

Proof. Similar to that of Lemma 5. QED.

Theorem 4. Reduction-completeness of F^* for normal forms. Let P be an F^* program and D_0 a term. Let D_0, D_1, \dots, D_n , $n \geq 0$, be a reduction in P , where D_n is in normal form. Then there is an NR-reduction $D_0, E_1, \dots, E_m = D_n$, $m \geq 0$, in P .

Proof. By induction on length n of D_0, D_1, \dots, D_n . If $n=0$, D_0 is already in normal form, so D_0 is the required NR-reduction.

Let $n > 0$ and assume theorem for D_1, \dots, D_n . Then there is an NR-reduction $D_1, F_2, \dots, F_p = D_n$. The situation can be laid out as follows:

$$\begin{array}{l}
 D_n \text{ (in normal form)} \\
 : \\
 | \\
 D_2 \\
 | \\
 D_1 \rightarrow F_2 \rightarrow \dots \rightarrow F_p = D_n \\
 | \\
 D_0 \rightarrow E_1 \rightarrow \dots \rightarrow E_m
 \end{array}$$

Since $D_0 \rightarrow D_1$ there are terms G, H , such that $G \Rightarrow H$ and $D_1 = D_0[G/H]$. Hence $R(G, H, D_0, D_1)$. Since D_1, F_2, \dots, F_p is an NR-reduction, by Lemma 6, there is an NR-reduction D_0, E_1, \dots, E_q such that $R(G, H, E_q, F_p)$. It is easily shown, by induction on length of terms, that there is an NR-reduction E_q, \dots, F_p . In each step in it, an occurrence of G is replaced by H . The required NR-reduction is then $D_0, E_1, \dots, E_q, \dots, F_p = D_n = E_m$. QED.

III. DETERMINISTIC F*

III.1 INTRODUCTION

A class of F* programs called **Deterministic F* (DF*)** is now defined and shown to possess several useful computational properties. In particular, every DF* program satisfies confluence and directedness. Confluence is shown to hold for any F* program which satisfies just restriction (f) below.

Confluence, means that if for terms M,N,P, $M \rightarrow^* N$, and $M \rightarrow^* P$, then there exists term Q such that $N \rightarrow^* Q$, and $P \rightarrow^* Q$. It has the immediate consequence that every term has at most one normal form. Hence DF* can be used as a functional programming system. Also, if a DF* program is interpreted as an equality theory, equality of two terms can be determined by checking whether their normal forms are syntactically identical.

Directedness, for simplified forms, means that if a term has a simplified form then *any* N-reduction starting at that term, if extended far enough, computes it. Moreover, all successful N-reductions are of equal length. Directedness, for normal forms, means that if a term has a normal form, then *any* NR-reduction starting at that term, if extended far enough, computes it. Moreover, all NR-reductions ending in normal forms are of equal length. Due to directedness, no searching among alternative N- or NR-reductions is necessary.

III.2 DEFINITION OF DF*

A DF* program is an F* program P satisfying two restrictions:

(f) Let LHS1 and LHS2 be variants of heads of two rules in P, such that LHS1 and LHS2 have no variables in common. Then LHS1 and LHS2 do not unify.

(g) Let $f(L_1, \dots, L_i, \dots, L_m) \Rightarrow \text{RHS}$ be a rule in P, where L_i is not a variable. Then in every other rule $f(K_1, \dots, K_i, \dots, K_m) \Rightarrow \text{RHS1}$ in P, K_i is not a variable.

Again, as can be seen from examples in this paper, and especially in Section VI, DF* is also quite expressive. Note that restrictions (a)-(e) are upon rules while (f) and (g) are upon the entire program. In the following, the first three rules do not constitute a DF* program because they violate (f) and the next four do not because they violate (g):

```
insert(A,[])=>[A].
insert(A,[UIV])=>[A,UIV].
insert(A,[UIV])=>[Uinsert(A,V)].
```

$f(X,[]) \Rightarrow []$.
 $f([],[U|V]) \Rightarrow []$.
 $a \Rightarrow a$.
 $b \Rightarrow []$.

Restriction (f), supported by (a)-(e), ensures confluence. Restriction (g), supported by (a)-(f), ensures directedness. In particular, given a term $f(t_1, \dots, t_i, \dots, t_n)$, it is possible to determine, at compile time, whether t_i needs to be simplified. It needs to be, only if the i th argument in the head of any F* rule defining f is a non-variable. Moreover, as shown below, the arguments of f which do need to be simplified can be simplified in any order.

The importance of select may be emphasized from the observation that even with restrictions (a)-(g), every outermost reduction strategy is not reduction complete. For example, given the DF* program:

$g([],X) \Rightarrow []$
 $g([U|V],W) \Rightarrow \dots$

and the term $E = g(b,a)$, a rightmost-outermost reduction strategy would compute the infinite reduction $g(b,a), g(b,a), \dots$. However, there exists a successful leftmost-outermost reduction $g(b,a), g([],a), []$. Select, of course, would compute this second reduction.

Thus, select is more than an outermost reduction strategy. For DF*, it may perhaps be called *outermost-call-by-need*. Note that it is still non-deterministic. However due to directedness, the non-determinism is benign.

$S_P(A,B)$. Let P be an F* program, and A, B be terms. Let G_1, \dots, G_m , $m \geq 0$, be mutually non-overlapping subterms in A , and H_1, \dots, H_m be terms such that for each $i \leq m$, $G_i \Rightarrow H_i$, and B is the result of *simultaneously* replacing G_1, \dots, G_m , in A , by respectively, H_1, \dots, H_m . Then we say $S_P(A,B)$. Note that G_1, \dots, G_m need not include all, or even one, of the mutually non-overlapping subterms of A which reduce as a whole. The subscript on S is dropped if clear from context.

$R@i$. Let R be an N-reduction E_0, E_1, \dots, E_m , $m \geq 0$, where for no i , $E_i \Rightarrow E_{i+1}$. Then there is some function symbol f , such that each E_i is of the form $f(p_1, \dots, p_k)$. Let $E_0 = f(t_1, \dots, t_k)$, and for any p , let R_p be E_0, E_1, \dots, E_p .

For any $1 \leq i \leq k$, $R_0@i$ is defined to be the singleton sequence t_i . For any $j \neq m$, let $E_j = f(a_1, \dots, a_{n-1}, a_n, a_{n+1}, \dots, a_k)$, and $E_{j+1} = f(a_1, \dots, a_{n-1}, b_n, a_{n+1}, \dots, a_k)$ such that a_n reduces to b_n in an N-step. If $n=i$, then $R_{j+1}@i = R_j@i : b_n$, otherwise, $R_{j+1}@i = R_j@i$. Here $:$ is concatenation of a term at the end of a sequence of terms.

For example, with the rules $a \Rightarrow a$, $b \Rightarrow b_1$, $b_1 \Rightarrow b_2$, and the N-reduction $R = f(a,b), f(a,b_1), f(a,b_1), f(a,b_2)$, $R@1 = a, a$, and $R@2 = b, b_1, b_2$. Thus, roughly, $R@i$ is the sequence, without duplicates, of i th arguments of the outermost function symbol of the members of the N-reduction R .

$R@u$. $R@u$ is a generalization of $R@i$ to positions in terms. Let R be an NR-reduction E_0, E_1, \dots, E_m , $m \geq 0$, where E_0 is simplified. Let A_0, A_1, \dots, A_p be unsimplified terms in E_0 such that no A_i is properly contained in any other unsimplified term. Let the positions of A_0, A_1, \dots, A_p in E_0 be u_1, \dots, u_p respectively. Then, for each $j \neq m$, E_{j+1} can be thought of as being derived from E_j by replacing a term A at one of the u_i by another term B . Moreover, A reduces to B in an NR-step.

For any u_i in u_1, \dots, u_p , $R_0@u_i$ is defined to be the singleton sequence A_i . For any j , $j \neq m$, let E_{j+1} be derived from E_j by replacing a term P at position u in E_j by Q , where u is in u_1, \dots, u_p . If $u = u_i$, $R_{j+1}@u_i = R_j@u_i:Q$, otherwise, $R_{j+1}@u_i = R_j@u_i$.

III.3 CONFLUENCE AND DIRECTEDNESS OF DF*

Confluence and directedness are shown by deriving the following results, for any DF* program P :

- (a) Let F_1, E_1, F_2 be terms such that $S(F_1, E_1)$, and $F_1 \rightarrow F_2$. Then there exists a term E_2 such that $E_1 \rightarrow^* E_2$, and $S(F_2, E_2)$.
- (b) If there are two N-reductions starting at the same term and ending in terms in simplified form, then these terms are identical, and the N-reductions are of equal length.
- (c) Let E_0, E_1, \dots, E_n be a successful N-reduction. Let $E_0 = F_0, F_1, \dots, F_p$, be an *unsuccessful* N-reduction, i.e. F_p is not simplified. Then $p < n$, and there exists F_{p+1} such that F_p reduces to F_{p+1} in an N-step.

Now, (a) is iterated to obtain confluence. (c) requires (b). From (c) we infer that if a term E_0 has a successful N-reduction then no N-reduction starting at E_0 is infinite, or terminates in failure. Hence, every N-reduction must terminate in a term in simplified form. Hence directedness for simplified forms. Similarly, directedness for normal forms.

Lemma 1. Select never chooses overlapping terms. Let P be a DF* program. Let E and F be terms such that $\text{select}(E, F)$. Then, for all G , $\text{select}(E, G)$ implies that G is not properly contained in F .

Proof. By induction on length of E. As before, the definition of select, for any term $f(T_1, \dots, T_n)$ is:

$\text{select}_P(f(T_1, \dots, T_n), f(T_1, \dots, T_n))$ if $f(T_1, \dots, T_n) \Rightarrow_P X$.
 $\text{select}_P(f(T_1, \dots, T_i, \dots, T_n), X)$ if
 there is a rule $f(L_1, \dots, L_i, \dots, L_n) \Rightarrow \text{RHS}$ in P, and
 there is no substitution σ such that $T_i = L_i \sigma$, and
 $\text{select}_P(T_i, X)$.

If E is a 0-ary function symbol, the lemma holds. Otherwise let $E = f(t_1, \dots, t_i, \dots, t_m)$ and let the lemma hold for each of t_1, \dots, t_m . Suppose $\text{select}(E, F)$ but $F \neq E$. Then for some t_i in t_1, \dots, t_m , $\text{select}(t_i, F)$. Suppose $\text{select}(E, G)$. If $G = E$ then G is clearly not contained in F. Otherwise, for some t_j in t_1, \dots, t_m , $\text{select}(t_j, G)$. If $j = i$, by induction hypothesis, G is not properly contained in F. If $j \neq i$, of course, G is not properly contained in F.

Suppose $\text{select}(E, F)$ and $F = E$. Then there exists a rule $f(M_1, \dots, M_i, \dots, M_m) \Rightarrow \text{RHS}$ such that E matches its head. Now suppose that there also exists G such that $\text{select}(E, G)$, and G is properly contained in F. Hence, for some t_i in t_1, \dots, t_m , $\text{select}(t_i, G)$. Hence there is a rule $f(L_1, \dots, L_i, \dots, L_m) \Rightarrow \text{RHS}_1$ such that t_i does not match L_i . Hence L_i is not a variable. By restriction (g) M_i is also not a variable. Hence t_i is in simplified form. But then $\text{select}(t_i, G)$ fails. Contradiction. QED.

Lemma 2. Let P be a DF* program. Let G be a term. Then there is at most one term H such that $G \Rightarrow H$.

Proof. By restriction (f). QED.

Lemma 3. Let σ and τ be two substitutions each defining only the variables X_1, \dots, X_m , $m \geq 0$, such that for any $i \leq m$, where $\langle X_i, s_i \rangle$ appears in σ , and $\langle X_i, t_i \rangle$ in τ , $S(s_i, t_i)$. Let M be a term, possibly containing variables, but only from X_1, \dots, X_m . Then $S(M\sigma, M\tau)$.

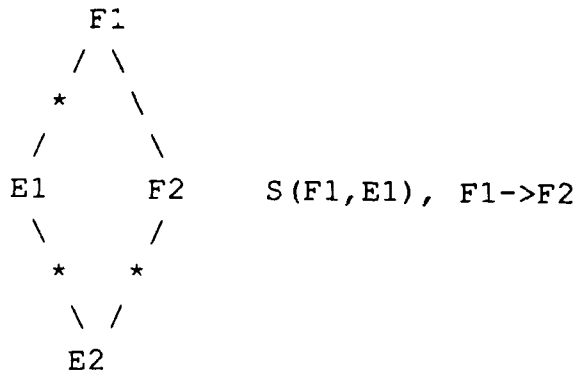
Proof. By induction on length of M. QED.

Lemma 4. Let P be a DF* program. Let $A = f(t_1, \dots, t_m)$, $m > 0$, and $B = f(t_1^*, \dots, t_m^*)$, such that for each $i \leq m$, $S(t_i, t_i^*)$. Let $A \Rightarrow C$. Then there exists D, such that $B \Rightarrow D$ and $S(C, D)$.

Proof. Let A reduce to C by the rule $\text{LHS} \Rightarrow \text{RHS}$. Then, there exists a substitution σ such that $A = \text{LHS}\sigma$ and $C = \text{RHS}\sigma$. It is easily verified that there exists substitution τ such that $B = \text{LHS}\tau$, σ and τ define the same variables, and for each i , where $\langle X_i, p_i \rangle$ appears in σ and $\langle X_i, q_i \rangle$ in τ , $S(p_i, q_i)$. Hence $B \Rightarrow \text{RHS}\tau = D$. By Lemma 3, $S(C, D)$. QED.

Lemma 5. Let P be a DF* program. Let $F1, E1, F2$ be terms such that $S(F1, E1)$, and $F1 \rightarrow F2$. Then there exists term $E2$ such that $E1 \rightarrow^* E2$, and $S(F2, E2)$.

Proof. By induction on length of $F1$. The situation can be visualized in the following diagram:



Case 1. $F1$ is a 0-ary function symbol. Since $F1 \rightarrow F2$, $F1 = F2$. If $F1 = E1$, take $E2 = F2$. Then $E1 = E2$, and $S(F2, E2)$, as required. Otherwise, $F1 = E1$. By restriction (f), $E1 = F2$. Take $E2 = F2$. Again, $E1 \rightarrow^* E2$, and $S(F2, E2)$, as required.

Case 2. $F1 = f(t1, \dots, tm)$, $m > 0$. Assume Lemma for $t1, \dots, tm$.

Case 2-1. $F1$ reduces to $E1$ as a whole. If $F1$ reduces to $F2$ as a whole, by restriction (f), $F2 = E1$. Take $E2 = E1$. Then $E1 \rightarrow^* E2$, and also $S(F2, E2)$, as required. Otherwise, there is some $j \leq m$, such that $tj \rightarrow tj^*$, and $F2 = f(t1, \dots, tj^*, \dots, tm)$. Hence $S(F1, F2)$. By Lemma 4, there exists $E2$ such that $F2 \rightarrow E2$ and $S(E1, E2)$. But then $E1 \rightarrow^* E2$. Since $F2 \rightarrow E2$, $S(F2, E2)$, as required.

Case 2-2. $F1$ does not reduce as a whole to $E1$. Then $E1 = f(s1, \dots, sm)$, and for each $i \leq m$, $S(ti, si)$.

If $F1 \rightarrow F2$, then, by Lemma 4, there exists $E2$ such that $E1 \rightarrow E2$, and $S(F2, E2)$, as required. Otherwise, there is some $j \leq m$, such that $tj \rightarrow tj^*$, and $F2 = f(t1, \dots, tj-1, tj^*, tj+1, \dots, tm)$. By induction hypothesis, there exists p , such that $sj \rightarrow^* p$ and $S(tj^*, p)$. Take $E2 = f(s1, \dots, sj-1, p, sj+1, \dots, sm)$. Clearly, $E1 \rightarrow^* E2$, and also, $S(F2, E2)$, as required. **QED.**

Lemma 6. Let P be a DF* program, and M, N, P be terms such that $S(M, N)$ and $M \rightarrow^* P$. Then there exists term Q such that $N \rightarrow^* Q$ and $S(P, Q)$.

Proof. By iterating Lemma 5. QED.

Lemma 7. Let P be a DF* program, and M, N, P be terms such that $M \rightarrow N$, and $M \multimap P$. Then there exists term Q such that $N \multimap Q$, and $P \multimap Q$.

Proof. Since $M \rightarrow N$, $S(M, N)$. By Lemma 6, there exists term Q such that $N \multimap Q$ and $S(P, Q)$. Hence $P \multimap Q$, as required. QED.

Theorem 1. Confluence of DF*. Let P be a DF* program, and M, N, P be terms such that $M \rightarrow N$, and $M \multimap P$. Then there exists term Q such that $N \multimap Q$, and $P \multimap Q$.

Proof. By iterating Lemma 7. QED.

Corollary. Uniqueness of normal forms. Let P be a DF* program. Then every term has at most one normal form.

Lemma 8. Let R be an N-reduction $f(p_1, \dots, p_k) = E_0, E_1, \dots, E_m$ such that for no i , $E_i \rightarrow E_{i+1}$. Then the length m of E_0, E_1, \dots, E_m is equal to the sum of the lengths of $R@1$, $R@2, \dots, R@k$.

Proof. By induction on m . If $m=0$, then clear. Assume lemma for E_0, \dots, E_{m-1} . There exists exactly one n , such that $E_{m-1} = f(t_1, \dots, t_{n-1}, t_n, t_{n+1}, \dots, t_k)$, $E_m = f(t_1, \dots, t_{n-1}, u_n, t_{n+1}, \dots, t_k)$, and t_n reduces to u_n in an N-step. So, for every i , $i \neq n$, $(E_0, \dots, E_{m-1})@i = (E_0, \dots, E_{m-1}, E_m)@i$. Only for n , $(E_0, \dots, E_{m-1}, E_m)@n = (E_0, \dots, E_{m-1})@n : u_n$. By induction hypothesis, the lemma is clear. QED.

Lemma 9. Let P be a DF* program. Let E_0, E_1, \dots, E_n be a successful N-reduction. Then for every successful N-reduction E_0, F_1, \dots, F_p , $p=n$ and $F_p = E_n$.

Proof: By induction on n . If $n=0$ then E_0 is simplified and the lemma holds trivially. Let $n>1$. Assume hypothesis for all successful N-reductions of length less than n .

Since E_0, E_1, \dots, E_n is a successful N-reduction, there exists E_k , $0 \leq k < n$, such that for no i , $0 \leq i < k$, $E_i \rightarrow E_{i+1}$, but $E_k \rightarrow E_{k+1}$, and E_{k+1}, \dots, E_n is a successful N-reduction. Let $E_0 = f(t_1, \dots, t_m)$, $m \geq 0$. Since $n > 0$, f is not a constructor symbol. Then $E_k = f(s_1, \dots, s_m)$ for terms s_1, \dots, s_m . Similarly, for the successful N-reduction F_0, F_1, \dots, F_p , there exists F_j with properties similar to those of E_k . The situation can be laid out in the following diagram:

$$E_0 = f(t_1, \dots, t_m) \text{ ---- } \multimap E_k = f(s_1, \dots, s_m) \Rightarrow E_{k+1} \text{ -- } \multimap E_n$$

$$F_0 = f(t_1, \dots, t_m) \text{ ---- } \multimap F_j = f(q_1, \dots, q_m) \Rightarrow F_{j+1} \text{ -- } \multimap F_p$$

Consider any t_i in t_1, \dots, t_m .

Case 1. t_i is simplified. Then, by definition of select, $t_i = s_i$. Similarly, $t_i = q_i$. Hence $q_i = s_i$.

Case 2. t_i is unsimplified. If s_i is simplified, then there exists a successful N-reduction $(E_0, \dots, E_k)@i$, of length less than n .

By restriction (g), the i th argument in the head of any rule in P , defining f , is a non-variable. Hence, q_i is also simplified. Hence there exists a successful N-reduction $(F_0, \dots, F_j)@i$. By induction hypothesis, its length is equal to that of $(E_0, \dots, E_k)@i$, and $q_i = s_i$.

If s_i is unsimplified, then, by restriction (g), and definition of select, $t_i = s_i = q_i$.

Hence $E_k = F_j$. By Lemma 8, the length of E_0, \dots, E_k is the sum of lengths of $(E_0, \dots, E_k)@i$ such that t_i is unsimplified but s_i is simplified. Again, by Lemma 8, and **Case 2**, this is also the length of F_0, \dots, F_j . Hence, $k = j$. By restriction (f), $E_{k+1} = F_{j+1}$. Now, E_{k+1}, \dots, E_n , is a successful N-reduction of length less than n . By induction hypothesis, its length is equal to that of the successful N-reduction F_{j+1}, \dots, F_p , and $E_n = F_p$. Hence, also, the length of E_0, \dots, E_n is equal to that of F_0, \dots, F_p . **QED.**

Lemma 10. Let P be a DF^* program. Let E_0, E_1, \dots, E_n and $E_0 = F_0, F_1, \dots, F_m$ be two NR-reductions such that E_n and F_m are in normal form. Then $E_n = F_m$ and $n = m$.

Proof: Note that $E_n = F_m$ follows directly from the confluence of DF^* . We focus on showing that $n = m$, and proceed by induction on length of E_0, E_1, \dots, E_n . If $n = 0$, then clear. Otherwise, let $n > 0$ and assume the lemma for NR-reductions of length less than n .

Case 1. E_0 is unsimplified. Then, there exists E_k , $0 < k \leq n$ such that E_0, \dots, E_k is a successful N-reduction. Also, there exists F_j , $0 < j \leq m$ such that E_0, \dots, F_j is a successful N-reduction. By Lemma 9, $F_j = E_k$ and $j = k$. Now E_k, \dots, E_n , and F_j, \dots, F_m are also NR-reductions. The length of E_k, \dots, E_n is less than n , so by induction hypothesis, $n = m$.

Case 2. E_0 is simplified. Then $E_0 = c(t_1, \dots, t_q)$ for terms t_1, \dots, t_q and constructor symbol c . If E_0 is in normal form, then the theorem holds. Otherwise, let A_0, A_1, \dots, A_p be unsimplified terms in E_0 such that no A_i is properly contained in any unsimplified term. Consider any A_i in A_0, A_1, \dots, A_p .

Let the position at which A_i occurs in E_0 be u_i . Then, since E_n is in normal form, $(E_0, \dots, E_n)@u_i$ is an NR-reduction ending in a normal form. Similarly obtain $(F_0, \dots, F_m)@u_i$.

By reasoning as in **Case 1**, the length of $(E_0, \dots, E_n)@u_i$ is equal to that of $(F_0, \dots, F_m)@u_i$. It can be shown, analogously to Lemma 8, that the length of E_0, \dots, E_n is equal to the sum of the lengths of each $(E_0, \dots, E_n)@u_i$. Similarly, for length of F_0, \dots, F_m . Hence $m=n$. **QED.**

Lemma 11. Let P be a DF^* program and E_0 a term. Let E_0, E_1, \dots, E_n be a successful N -reduction. Let $E_0 = F_0, F_1, \dots, F_p$, be an *unsuccessful* N -reduction, i.e. F_p is not simplified. Then $p < n$, and there exists F_{p+1} such that F_p reduces to F_{p+1} in an N -step.

Proof: By induction on the length of E_0, \dots, E_n . If $n=0$ then E_0 is in simplified form and the only N -reduction starting at E_0 is E_0 itself, so the lemma is clear. Let $n > 0$. Then E_0 is not simplified. Assume lemma for all successful N -reductions of length less than n . Since E_0, E_1, \dots, E_n is a successful N -reduction, there exists E_k , $0 \leq k < n$ such that for no i , $0 \leq i < k$, $E_i \Rightarrow E_{i+1}$, but $E_k \Rightarrow E_{k+1}$, and E_{k+1}, \dots, E_n is a successful N -reduction. Let $E_0 = f(t_1, \dots, t_m)$, $m \geq 0$. Let $E_k = f(s_1, \dots, s_m)$. We have two cases:

Case 1. There exists F_j , $0 \leq j < p$ such that for no i , $0 \leq i < j$, $F_i \Rightarrow F_{i+1}$, but $F_j \Rightarrow F_{j+1}$, and F_{j+1}, \dots, F_p is an N -reduction. Let $F_j = f(q_1, \dots, q_m)$. The situation can be visualized in the following diagram:

$$E_0 = f(t_1, \dots, t_m) \dashrightarrow^* E_k = f(s_1, \dots, s_m) \Rightarrow E_{k+1} \dashrightarrow^* E_n$$

$$F_0 = f(t_1, \dots, t_m) \dashrightarrow^* F_j = f(q_1, \dots, q_m) \Rightarrow F_{j+1} \dashrightarrow^* F_p$$

By reasoning as in Lemma 9 above, $E_k = F_j$, and $k=j$. By restriction (f), $E_{k+1} = F_{j+1}$. By induction hypothesis, $p < n$. Furthermore, there exists F_{p+1} such that F_p reduces to F_{p+1} in an N -step.

Case 2. There does not exist F_j in F_0, \dots, F_p such that $F_j \Rightarrow F_{j+1}$. Let $F_p = f(q_1, \dots, q_m)$. The situation can be visualized as:

$$E_0 = f(t_1, \dots, t_m) \dashrightarrow^* E_k = f(s_1, \dots, s_m) \Rightarrow E_{k+1} \dashrightarrow^* E_n$$

$$F_0 = f(t_1, \dots, t_m) \dashrightarrow^* F_p = f(q_1, \dots, q_m) \quad (\text{unsimplified})$$

It is easily seen that either $F_p \Rightarrow F_{p+1}$, or there exists i in $1, \dots, m$ such that q_i is not simplified, but s_i is. By induction hypothesis, there exists r_i such that q_i reduces to r_i in an N -step. Hence F_p reduces to $f(q_1, \dots, r_i, \dots, q_m)$ in an N -step. Also, by Lemma 8, and induction hypothesis, $p < n$. **QED.**

Theorem 2. Directedness for simplified forms. Let P be a DF^* program. Let E_0

be a term and let E_0, \dots, E_n be a successful reduction. Then any N-reduction starting at E_0 , if extended far enough, would terminate in a term in simplified form.

Proof. By reduction-completeness for simplified forms, (Theorem 2, Section II), and iterating Lemma 11. QED.

Lemma 12. Let P be a DF* program and E_0 a term. Let E_0, E_1, \dots, E_n be an NR-reduction such that E_n is in normal form. Let $E_0 = F_0, F_1, \dots, F_p$, be an NR-reduction such that F_p is not in normal form. Then, $p < n$, and there exists F_{p+1} such that F_p reduces to F_{p+1} in an NR-step.

Proof: By induction on length of E_0, E_1, \dots, E_n . If $n=0$, then clear. Otherwise, assume Lemma for all NR-reductions of length less than n , and ending in normal forms.

Case 1. E_0 is unsimplified. By a reasoning very similar to that in proof of Lemma 11.

Case 2. E_0 is simplified. If E_0 is in normal form, the lemma trivially holds. Otherwise, as in Case 2 of Lemma 10. QED.

Theorem 3. Directedness for normal forms. Let P be a DF* program. Let E_0 be a term and let E_0, \dots, E_n be a reduction where E_n is in normal form. Then any NR-reduction starting at E_0 , if extended far enough, would terminate in a normal form.

Proof By reduction-completeness for normal forms, (Theorem 4, Section II), and iterating Lemma 12. QED.

IV. LABELED DETERMINISTIC F*

IV.1 INTRODUCTION

Intuitively, it can be seen that in an N-reduction a term is reduced only when it is necessary for simplifying the first term in the reduction. In this sense, an N-reduction conserves computation. For example, with the rules:

$$\begin{aligned}f(X) &\Rightarrow [X]. \\ a &\Rightarrow [].\end{aligned}$$

there exists the N-reduction $f(a), [a]$. There is no N-reduction starting at $f(a)$ in which a is reduced. However, it can also happen that in an N-reduction, several copies of the same term are reduced. This can happen when use is made of a rule in which a variable occurs more than once on the right hand side. In this sense, an N-reduction wastes computation. For example, with the rules:

$$\begin{aligned}f(X) &\Rightarrow g(X, X). \\ g([], []) &\Rightarrow []. \\ a &\Rightarrow [].\end{aligned}$$

there exists the N-reduction $f(a), g(a, a), g([], a), g([], []), []$. The two occurrences of a in the second term are copies of each other, yet they are reduced separately. This is the same problem which arises with a call-by-name procedure call mechanism in programming languages.

If we can arrange that when a term is reduced, all copies of it are also reduced, then N-reductions could become considerably shorter. In fact, it is shown that they become *minimal*. An N-step in which all copies of a term are replaced is called an NA-step. Thus it is distinguished from an N-step in which only a single copy of a term is replaced. A sequence of NA-steps is called an NA-derivation. The prefix NA stands for "normal-all". Minimality yields a strong form of laziness, since terms are simplified with minimum computational effort.

The notion of a copy of a term, however, has two legitimate interpretations. The first is simply that any two occurrences of a subterm in a term are copies of each other.

The second is obtained from examining representations of terms as directed acyclic graphs. A 0-ary function symbol f is represented as a graph consisting of just a single node with f stored in it. A term $f(t_1, \dots, t_n)$ is represented as a graph whose root is a node with $n+1$ fields. The first field stores f and for each $1 \leq i \leq n$, the i th field stores a pointer

to the graph representation of t_i . A term can have many graph representations. For example, the two occurrences of a in $f(a,a)$ can be represented by a single graph, or by distinct graphs. Now, two occurrences of a subterm in term E are said to be copies of each other *only* if, in the graph representation of E , they have the same graph representation.

We adopt the second interpretation since it enables us to develop a simple proof of minimality and also to implement replacement of all copies of a subterm with a small overhead. Graph representations of terms are, in turn, represented using labeled terms. The address of each node in a graph is represented by a label. Let there be a graph G with root node N . Let N contain $m+1$ fields, where the first field contains the symbol f and the rest of the m fields contain pointers to, respectively, graphs G_1, \dots, G_m . Let the address of N be represented by label α . Then the representation of G is the labeled term $f(\alpha, t_1, \dots, t_m)$ where for each i , the representation of G_i is the labeled term t_i .

A subset of DF^* called, Labeled Deterministic F^* (LDF^*), is defined. Notions of labels [Vuillemin 1974], labeled terms, ordinary terms, and ordinary programs are introduced. Reductions in LDF^* are intended to mimic graph reduction. In particular, it is ensured that when a new node is allocated in graph reduction, a label not previously used in the LDF^* reduction is generated.

LDF^* is shown to be minimal in the following sense: where P^* is an LDF^* program, and E a proper term, let there be a shortest successful reduction of E . Then there is a successful NA-derivation of E of lesser or equal length.

It is also shown that with each ordinary DF^* program P , one can associate an LDF^* program P^* , such that if there is a successful reduction in P , there is a successful reduction in P^* of exactly equal length. Hence, to simplify terms in P in a minimum number of steps, it is sufficient to transform P to P^* and use NA-derivations.

Some main ideas in our proof are (a) in each reduction step, a label is eliminated, so (b) the size of the elimination-set (E-set) of a reduction, i.e. the set of labels eliminated in the reduction, is a lower-bound on its length, (c) the size of the E-set of an NA-derivation of a proper term, is exactly equal to its length.

IV.2 DEFINITION OF LDF^*

Labels. Let $\alpha, \beta, \xi, \alpha_1, \beta_1, \xi_1, \dots$ be an enumerably infinite subset of the set of 0-ary function symbols in F^* . Each member of this list is called a primitive label.

Let $*$ be a binary function symbol in F^* . A label is defined as follows. A primitive label

is a label. If x and y are labels then $x*y$ is also a label. A label α is said to be a proper initial segment of label β if either $\beta=\alpha*\delta$, or $\beta=\xi*\delta$ and α is a proper initial segment of ξ .

Labeled terms. Where f is an $n+1$ -ary function symbol, $n \geq 0$, $f \neq *$, α a label and t_1, \dots, t_n labeled terms, $f(\alpha, t_1, \dots, t_n)$ is a labeled term. α is called the outermost label of $f(\alpha, t_1, \dots, t_n)$.

For example, where f is a 4-ary function symbol and a, b are 1-ary function symbols, $f(\delta, a(\alpha), b(\beta), a(\xi))$ is a labeled term, and δ is the outermost label of this term. Note that a label standing alone is *not* a labeled term. Neither is a labeled term of the form $A*B$. *Also, note that a labeled term never contains any variables.*

A labeled term is said to be in normal form if it contains only constructor symbols and labels.

Maximal labels. A label is maximal in a labeled term if it is not a proper initial segment of any other label in that term.

Proper terms. A labeled term E is called proper if (a) all its labels are maximal, and (b) for every two subterms A and B of E , if A and B have the same outermost label then $A=B$. For example, $f(\alpha, b(\xi), c(\delta))$ is a proper term. However, $g(\alpha, a(\alpha*\delta))$ is not a proper term since it violates (a), and $f(\alpha, b(\alpha))$ is not a proper term since it violates (b).

Ordinary terms, ordinary rules, and ordinary programs. A term in F^* , possibly containing variables, a rule in F^* , or an F^* program, is said to be ordinary if it does not contain any labels, nor any occurrence of the symbol $*$.

A mapping Σ . Let F be the set of all function symbols in F^* , except $*$, and the primitive labels. Let there be an injection Σ between F and F which maps each n -ary function symbol in F to an $n+1$ -ary function symbol in F . Moreover, Σ always maps a constructor symbol to a constructor symbol, and a non-constructor symbol to a non-constructor symbol.

Labeled versions of ordinary terms, possibly containing variables. Let E be an ordinary term, possibly containing variables. If E is a variable then its labeled version is E itself. Otherwise, let $E=f(t_1, \dots, t_n)$, $n \geq 0$. Its labeled version is $f_{\alpha}(\alpha, t_{1_{\alpha}}, \dots, t_{n_{\alpha}})$ where α is a label, $\Sigma(f)=f_{\alpha}$, and for each i , $t_{i_{\alpha}}$ is a labeled version of t_i . For example, where Σ maps f to f_{α} and a to a_{α} , a labeled version of $f(a, a)$ is $f_{\alpha}(\alpha, a_{\alpha}(\beta), a_{\alpha}(\delta))$.

Labeled versions of ordinary rules. Let $LHS \Rightarrow RHS$ be an ordinary F^* rule. A labeled version of this rule, $LHS^* \Rightarrow RHS^*$, is defined as follows:

Let $LHS = f(L_1, \dots, L_m)$. Then $LHS^* = f(L, L_1, \dots, L_m)$ satisfying the following conditions: (a) $f = \Sigma(f)$, (b) L is a variable, (c) If L_i is a variable, $L_i = L_i$, otherwise $L_i = c(X_1, \dots, X_m)$, and $L_i = c(K_i, X_1, \dots, X_m)$, K_i a variable, $\Sigma(c) = c$, and (d) a variable occurs at most once in LHS^* .

Let RHS_1 be a labeled version of RHS in which all labels are distinct, and for no two of these is one a proper initial segment of the other. Let these labels be β_1, \dots, β_k . Then, where $LHS^* = f(L, L_1, \dots, L_m)$, RHS^* is obtained by replacing, in RHS_1 , each β_i by $L^*\beta_i$.

Labeled Deterministic F* Programs. Let P be an ordinary DF* program, and let P^* consist of labeled versions of rules in P . Then P^* is called a labeled deterministic F* (LDF*) program. For example, where P consists of:

$$\begin{aligned} \text{append}(\text{nil}, X) &\Rightarrow X \\ \text{append}(\text{cons}(U, V), W) &\Rightarrow \text{cons}(U, \text{append}(V, W)) \end{aligned}$$

and Σ maps append , nil , and cons to $\text{append}_$, $\text{nil}_$ and $\text{cons}_$ respectively, P^* consists of:

$$\begin{aligned} \text{append}_(L, \text{nil}_(L_1), X) &\Rightarrow X. \\ \text{append}_(L, \text{cons}_(K_1, U, V), W) &\Rightarrow \text{cons}_(L^*\beta_1, U, \text{append}_(L^*\beta_2, V, W)) \end{aligned}$$

where β_1 , and β_2 are distinct labels, and neither is a proper initial segment of each other. Note that each LDF* program is a DF* program as well as an F* program. Also, each labeled term is an F* term. *Hence, results of all previous sections also hold for LDF* programs and labeled terms.*

NA-steps and NA-derivations. Let P be an LDF* program and E, G, H be labeled terms. Suppose $\text{select}_P(E, G)$ and $G \Rightarrow_P H$. Let E_1 be the result of replacing *all* occurrences of G by H in E . Then we say that E reduces to E_1 in an NA-step in P . The prefix NA in N-step stands for "normal-all". A sequence of labeled terms E_0, E_1, \dots is an **NA-derivation** if for each i , when E_i and E_{i+1} both exist, E_i reduces to E_{i+1} in an NA-step. For example, given the rule $a(L) \Rightarrow b(L^*\alpha)$, the term $f(\beta, a(\delta), a(\delta))$ reduces in an NA-step to $f(\beta, b(\delta^*\alpha), b(\delta^*\alpha))$.

Leftmost steps and reductions. Let P be an F* program, not necessarily ordinary. Let E be a term, and G and G_1 two of its subterms. G is said to be to the left of G_1 in E , if either (a) they both occur at the same position in E , or (b) in the depth-first or preorder traversal of the tree representation of E , the function symbol which is the root of G occurs before the function symbol which is the root of G_1 .

Let $\text{select}(E, G)$, $G \Rightarrow H$. Then E reduces to F in a leftmost N-step, if for every G_1 ,

$\text{select}(E, G1)$ implies G is to the left of $G1$ in E , and $F = E[G/H]$.

Let $\text{select}(E, G)$, $G \Rightarrow H$. Then E reduces to F in a leftmost NA-step, if for every $G1$, $\text{select}(E, G1)$ implies G is to the left of $G1$ in E , and F is the result of replacing all occurrences of G in E by H .

Definitions of leftmost N-reductions and leftmost NA-derivations are the obvious ones.

Elimination sets or E-sets. Let A_0, A_1, \dots, A_n be a reduction where each A_j is a labeled term. For any i , let $A_{i+1} = A_i[G/H]$ where $G = f(\alpha, t_1, \dots, t_m)$. Then we say that the function-label pair (FL-pair) $\langle f, \alpha \rangle$ has been eliminated in the reduction A_i, A_{i+1} . The elimination set or E-set of a reduction is defined as the set of all FL-pairs eliminated in the reduction. Since elimination of an FL-pair requires one reduction step, the size of this set (number of elements in it) is a lower bound on the length of the reduction (the number of steps in it).

Since an NA-step can be thought of as a sequence of reductions steps, an NA-derivation can be thought of as a reduction. Hence, the E-set of an NA-derivation is the E-set of the corresponding reduction.

IV.3 MINIMALITY OF LDF*

Let P be an LDF* program and E a labeled term. We already know from completeness of DF* that if E has a successful reduction, it has a successful N-reduction. By directedness of DF*, it has a successful leftmost N-reduction. We now show the following:

- (a) If E has a successful reduction R_0 , E has a successful N-reduction R_1 whose E-set is a subset of the E-set of R_0 .
- (b) If R_1 and R_2 are two successful N-reductions of E , their E-sets are identical.
- (c) If E has a successful leftmost N-reduction R_2 , it has a successful leftmost NA-derivation R_3 . Furthermore, the E-set of R_3 is a subset of the E-set of R_2 .
- (d) If E is a proper term, the size of the E-set of any NA-derivation starting at E is equal to the number of NA-steps in that derivation.
- (e) Let P be an ordinary DF* program, and P^* its labeled version. Let E_0 be an ordinary term, and E_0^* a labeled version of E_0 . Let E_0, E_1, E_2, \dots be a reduction in P . Then there exists a reduction $E_0^*, E_1^*, E_2^*, \dots$ in P^* , such that for each i , E_i^* is a labeled version of E_i .

Let E be a proper term. Let R_0 be a shortest successful reduction of E . Then its length is greater than or equal to the size of its E-set. By (a), there exists R_1 , an N-reduction of E whose E-set is a subset of that of R_0 . By directedness of DF^* , there exists R_2 , a successful leftmost N-reduction of E . By (b), the E-set of R_1 is identical to that of R_2 . By (c), there exists R_3 , a successful leftmost NA-derivation of E whose E-set is a subset of that of R_2 . By (d), the length of this NA-derivation is at most the length of R_0 . Hence, leftmost NA-derivations are minimal for simplifying proper terms.

Now, let P be an ordinary DF^* program and P^* its labeled version. Let there be a successful reduction R of a term E in P . By (e) there exists a successful reduction R^* of a labeled version E^* of E . This version can always be chosen to be proper. By minimality of LDF^* , there is a successful NA-derivation starting at E^* of length less than or equal to that of R^* or R . Hence, to simplify terms in a minimum number of steps, it is sufficient to transform P to P^* and use leftmost NA-derivations. This reasoning is now carried out formally and in detail.

IV.3.1 Existence of successful leftmost NA-derivations

Lemma 1. Let P be an LDF^* program. Let E_0, F_0 be labeled terms such that $E_0 \rightarrow F_0$. If E_0 reduces to E_1 in a leftmost N-step then there exists F_1 such that $E_1 \rightarrow^* F_1$, and (a) either $F_1 = F_0$, or (b) F_0 reduces to F_1 in a leftmost N-step, and the FL-pair eliminated in F_0, F_1 is the same as that eliminated in E_0, E_1 .

Proof. By induction on length of E_0 . We can draw the following diagram:

$E_0 \text{-----} \rightarrow$	F_0
leftmost	$F_0 = F_1$ or
N-step	F_0 reduces to F_1 in a leftmost N-step
$E_1 \text{-----} \rightarrow^*$	F_1

Case 1. $E_0 = f(\alpha)$ for some 1-ary function symbol f and label α . Since $E_0 \rightarrow F_0$, $E_0 \Rightarrow F_0$. So, $\text{select}(E_0, E_0)$ and due to restriction (f), $E_1 = F_0$. Take $F_1 = F_0$. Clearly, $E_1 \rightarrow^* F_1$.

Case 2. $E_0 = f(\alpha, t_1, \dots, t_m)$, for some $m+1$ -ary function symbol f , $m > 0$, label α , and labeled terms t_1, \dots, t_m .

Case 2-1. $E_0 \Rightarrow F_0$. Similar to Case 1. $E_1 = F_1 = F_0$ and $E_1 \rightarrow^* F_1$.

Case 2-2. Not $E_0 \Rightarrow F_0$. Then $F_0 = f(\alpha, t_1^*, \dots, t_m^*)$, and there exists j such that t_j -

$\rightarrow t_j^*$ and for each $i, i \neq j$ implies $t_i = t_i^*$.

Suppose $E_0 \Rightarrow E_1$. Then the FL-pair eliminated in E_0, E_1 is $\langle f, \alpha \rangle$. It is easily verified by induction on length of E_0 , that $F_0 \Rightarrow F_1$ and $E_1 \rightarrow F_1$. Also, F_0 reduces to F_1 in a leftmost N-step. Finally, the FL-pair eliminated in F_0, F_1 is also $\langle f, \alpha \rangle$.

Suppose not $E_0 \Rightarrow E_1$. Then, there is some k , such that $E_1 = f(t_1, \dots, t_{k-1}, sk, t_{k+1}, \dots, t_m)$, and t_k reduces to sk in a leftmost N-step. By induction hypothesis, there exists sk^* such that $sk \rightarrow sk^*$, and either $t_k^* = sk^*$, or t_k^* reduces to sk^* in a leftmost N-step, and the FL-pair eliminated in t_k, sk is the same as that eliminated in t_k^*, sk^* .

If $t_k^* = sk^*$, let $F_1 = F_0$. Clearly, $E_1 \rightarrow F_1$. Otherwise, let $F_1 = f(\alpha, t_1^*, \dots, t_{k-1}^*, sk^*, t_{k+1}^*, \dots, t_m^*)$. Since t_k^* reduces to sk^* in an N-step, t_k^* is not simplified. Hence, using restriction (g), it is easily verified that F_0 reduces to F_1 in a leftmost N-step. In particular, in this step, t_k^* reduces to sk^* in a leftmost N-step. Hence, $E_1 \rightarrow F_1$, and the FL-pair eliminated in E_0, E_1 is the same as that eliminated in F_0, F_1 . **QED.**

Lemma 2. Let P be an LDF* program. Let E_0, F_0 be labeled terms such that $E_0 \rightarrow F_0$. If E_0 reduces to E_1 in a leftmost N-step then there exists F_1 such that $E_1 \rightarrow F_1$, and (a) either $F_1 = F_0$, or (b) F_0 reduces to F_1 in a leftmost N-step, and the FL-pair eliminated in F_0, F_1 is the same as that eliminated in E_0, E_1 .

Proof. By induction on length of the reduction E_0, \dots, F_0 , and using Lemma 1. **QED.**

Lemma 3. Let P be an LDF* program. Let E_0, F_0 be labeled terms such that $E_0 \rightarrow F_0$ and let there be a successful leftmost N-reduction E_0, E_1, \dots, E_n . Then there is a successful leftmost NA-derivation $F_0, F_1, F_2, \dots, F_k$ whose E-set is a subset of that of E_0, E_1, \dots, E_n .

Proof. By induction on length n of E_0, \dots, E_n , and using Lemma 2. **QED.**

Theorem 1. Let P be an LDF* program. Let E_0 be a labeled term and E_0, E_1, \dots, E_n a successful leftmost N-reduction. Then there is a successful leftmost NA-derivation $E_0, F_1, F_2, \dots, F_k$ whose E-set is a subset of that of E_0, E_1, \dots, E_n .

Proof. Since $E_0 \rightarrow E_0$, apply Lemma 3. **QED.**

IV.3.2 E-sets of N-reductions

Lemma 4. Let P be an LDF* program and $E1, F1, G, H$ be labeled terms such that:

- (a) $R(G, H, E1, F1)$, and
- (b) $F1$ reduces to $F2$ in an N-step, and
- (c) The outermost function symbol and label of G are, respectively, g and α , and
- (d) $\langle r, \beta \rangle$ is the FL-pair eliminated in the reduction $F1, F2$.

Then there exists an N-reduction $E1, \dots, E2$ such that its E-set is included in $\{\langle g, \alpha \rangle, \langle r, \beta \rangle\}$ and $R(G, H, E2, F2)$.

Proof. Exactly parallel to proof of Theorem 1, Section II. **QED.**

Lemma 5. Let P be an LDF* program. Let $E1, F1, G, H$, be labeled terms such that $R(G, H, E1, F1)$. Let the outermost function symbol and label of G be g and α respectively. Let $F1, F2, \dots, Fm$ be a successful N-reduction. Then there exists a successful N-reduction $E1, \dots, En$ whose E-set is contained in the union of $\{\langle g, \alpha \rangle\}$ and the E-set of $F1, F2, \dots, Fm$.

Proof. By induction on length of $F1, \dots, Fm$. **QED.**

Lemma 6. Let $E1, F1, G2, \dots, Gm$ be a successful reduction. Then there is a successful N-reduction $E1, \dots, En$ such that the E-set of $E1, \dots, En$ is contained in that of $E1, F1, G2, \dots, Gm$.

Proof. By induction on length of $E1, F1, G2, \dots, Gm$, and Lemma 5. **QED.**

Lemma 7. Let P be an LDF* program. Let $E0$ be a labeled term and $E0, E1, \dots, En$ and $E0, F1, \dots, Fp$ two successful N-reductions. Then, the E-set of one is identical to that of the other.

Proof. Exactly analogous to the proof of Lemma 9, Section III, that any two successful N-reductions of a term are of equal length, and end in the same simplified form. **QED.**

IV.3.3 Reductions of proper terms

Lemma 8. Let P be an LDF* program. Let E be a proper term and let E reduce to F in an NA-step. Then all labels of F are maximal.

Proof. Let $\text{select}(E, G)$. Then $G \Rightarrow H$ and F is obtained by replacing all occurrences of G in E by H . Let the rule by which $G \Rightarrow H$ be $\text{LHS} \Rightarrow \text{RHS}$, and let $G = g(\alpha, t1, \dots, tm)$, $m \geq 0$,

and each t_i a labeled term. Take any two labels β and ξ in F . There are four cases.

Case 1. β and ξ are both in E . Since E is proper, these labels are not proper initial segments of each other.

Case 2. Only β is in E . Then, by the nature of LDF* rules, $\xi = \alpha * \delta$ for some label δ in RHS. Since E is proper, β and α are not proper initial segments of each other. If $\beta \neq \alpha$ then β and $\alpha * \delta$ are also not proper initial segments of each other.

Suppose $\beta = \alpha$. Since β occurs in E , E has a subterm $f(\beta, s_1, \dots, s_n)$, $n \geq 0$. Since E is proper, $f(\beta, s_1, \dots, s_n) = G$. But since all occurrences of G are replaced by H , β cannot occur in F , as assumed. Hence this subsubcase cannot arise.

Case 3. Only ξ is in E . Same as case 2.

Case 4. None of β and ξ is in E . Then, by definition of LDF* rules, $\beta = \alpha * \delta$ and $\xi = \alpha * \epsilon$, for some labels δ and ϵ in RHS. Since δ and ϵ are not proper initial segments of each other, neither are β and ξ . **QED.**

Lemma 9. Let P be an LDF* program. Let E be a proper term and let E reduce to F in an NA-step. Let A and B be two subterms of F such that the outermost label of A and of B is β . Then, $A=B$.

Proof. Let $\text{select}(E, G)$. Then $G \Rightarrow H$ and F is obtained by replacing all occurrences of G in E by H . Let the rule by which $G \Rightarrow H$ be LHS \Rightarrow RHS, and let the outermost label of G be α . There are four cases.

Case 1. A , but not B , is a subterm of H . Let the label of A , and of B be β . Since B is not a subterm of H , there occurs B_1 in E , with label β , such that B is the result of replacing all occurrences of G in B_1 by H .

Since A is a subterm of H , β occurs in H . However, $\beta \neq \alpha * \delta$ since α , and β both occur in E , and E is proper. Hence, β occurs in G . But then, since E is proper, B cannot properly contain G . Hence $B_1=B$, so B occurs in E .

Now β is also the label of A , $\beta \neq \alpha * \delta$, and A occurs in H . Hence A occurs in G , and so in E . Since E is proper, $A=B$, as required.

Case 2. B , but not A , is a subterm of H . Then, as in the previous case, B occurs in E . Hence $A=B$.

Case 3. Both A and B are subterms of H. Then A and B are also contained in G. Suppose A is not, but B is. Then, $\beta = \alpha * \delta$. Since B occurs in G, β also occurs in E. Contradiction with E is proper. Similarly, for A in G, but not B. Suppose none of A and B are in G. Without loss of generality assume A and B occur at distinct positions. Then, there must be distinct labels ϵ and ϕ in RHS such that $\beta = \alpha * \epsilon$ and $\beta = \alpha * \phi$. But this implies $\epsilon = \phi$ which, by the nature of labeled rules, is impossible. Hence both A and B are contained in G, and hence in E. Since E is proper, $A = B$.

Case 4. None of A and B is a subterm of H. Hence, there exist terms A1 and B1 in E such that A is obtained by replacing all occurrences of G in A1 by H and B is obtained from B1 similarly. Since $A \neq H$, $B \neq H$, the outermost label of A1 and B1 is also β . Since E is proper $A1 = B1$. Hence $A = B$. QED.

Lemma 10. Let P be an LDF* program. Let a proper term E reduce to F in an NA-step. Then F is a proper term.

Proof. By Lemmas 8 and 9, F is a proper term. QED.

Lemma 11. Let P be an LDF* program. Let E0 be a proper term. Let $E0, E1, \dots, Ek$ be an NA-derivation. Then, in this reduction, an FL-pair is eliminated at most once.

Proof. By induction on length k of $E0, E1, \dots, Ek$. If $k=0$, then clear. Otherwise, assume the theorem for $E1, \dots, Ek$. Let $\text{select}(E0, G)$, $G \Rightarrow H$ and let E1 be obtained by replacing all occurrences of G in E0 by H. Let the outermost function symbol of G be f and its outermost label be α . Hence, $\langle f, \alpha \rangle$ is the pair eliminated in the reduction $E0, E1$.

If we can show that there is no term $f(\alpha, t1, \dots, tm)$ in $E1, \dots, Ek$, then, by induction hypothesis, we can conclude that no FL-pair is eliminated more than once in $E0, E1, \dots, Ek$. To show this, it is sufficient to show that the label α never occurs in $E1, \dots, Ek$. Since E0 is proper, and $\langle f, \alpha \rangle$ is eliminated, α does not occur in E1. Let ξ be a label in $E2, \dots, Ek$. Then, either $\xi = \beta$ for some label β in E1 in which case $\xi \neq \alpha$. Otherwise, $\xi = \beta * \epsilon$ for some label β in E1 and label ϵ . We show that it is not possible that $\beta * \epsilon = \alpha$.

Case 1. β occurs in E0. Since E0 is proper, β is not a proper initial segment of α . Hence, it is not possible that $\beta * \epsilon = \alpha$.

Case 2. β does not occur in E0. Then, by the nature of LDF* rules, $\beta = \alpha * \delta$, for some label δ . Hence, $\beta * \epsilon$ is longer than α , and so $\beta * \epsilon \neq \alpha$. QED.

Theorem 2. Minimality of LDF*. Let P be an LDF* program. Let E0 be a proper term. Let $E0, E1, \dots, Ek$ be a successful reduction. Then there exists a successful leftmost NA-

derivation E_0, F_1, \dots, F_m such that $m \leq k$.

Proof. Since E_0, E_1, \dots, E_k is a successful reduction, by reduction-completeness for F^* , Theorem 2, Section II, there exists a successful N-reduction E_0, G_1, \dots, G_n . Let the E-set of E_0, E_1, \dots, E_k be S_1 and that of E_0, G_1, \dots, G_n be S_2 . Then, by Lemma 6, S_2 is a subset of S_1 . By directionality of DF^* , there exists a successful leftmost N-reduction E_0, H_1, \dots, H_n . By Lemma 7, its E-set is also S_2 .

By Theorem 1 above, there exists a successful leftmost NA-derivation E_0, F_1, \dots, F_m whose E-set, S_3 , is a subset of S_2 . Hence S_3 is a subset of S_1 . By Lemma 11, the size of S_3 is m . The size of S_1 is a lower bound on the number of steps in E_0, E_1, \dots, E_k . Hence $m \leq k$. QED.

IV.4 EXTENSION OF MINIMALITY RESULT TO NORMAL FORMS

We have shown that leftmost NA-derivations reduce proper terms to simplified forms in a minimum number of steps. It appears to be straightforward to extend this result to normal forms.

E reduces to F in an NAR-step if $\text{select-r}(E, p)$, $p \Rightarrow q$ and F is the result of replacing each occurrence of p in E by q . Definitions of NAR-derivations and leftmost NAR-derivations are the obvious ones. The proof that leftmost NAR-reductions reduce proper terms to normal forms in a minimum number of steps appears to be very similar to the above proof.

IV.5 DERIVED MINIMALITY OF DF^*

Lemma 12. Let P be a DF^* program and E_0 a term, where both P and E_0 are ordinary. Let P^* and E_0^* be, respectively, their labeled versions. Let $E_0 \xrightarrow{P} E_1$. Then there exists E_1^* such that $E_0^* \xrightarrow{P^*} E_1^*$ and E_1^* is a labeled version of E_1 .

Proof. There exist G, H such that $E_1 = E_0[G/H]$. Proceed by induction on length of E_0 . If E_0 is a 0-ary function symbol g , then clear.

Otherwise, $E_0 = f(t_1, \dots, t_m)$, $m > 0$. Then $E_0^* = f^*(\alpha, t_1^*, \dots, t_m^*)$ where $\Sigma(f) = f^*$ and for each i , $0 \leq i \leq m$, t_i^* is a labeled version of t_i . Assume lemma for each of t_1, \dots, t_m .

Suppose G occurs in some t_i in t_1, \dots, t_m , $d_i = t_i[G/H]$ and $E_1 = f(t_1, \dots, t_{i-1}, d_i, t_{i+1}, \dots, t_m)$. Then, by induction hypothesis, there exists d_i^* such that $t_i^* \xrightarrow{G} d_i^*$ and d_i^* is a labeled version of d_i . Let $E_1^* = f^*(\alpha, t_1^*, \dots, t_{i-1}^*, d_i^*, t_{i+1}^*, \dots, t_m^*)$. Clearly, E_1^* is a labeled version of E_1 .

Suppose $G=E0$. Then there is a rule $f(L1,...,Lm)=>RHS$ such that $E0$ matches $f(L1,...,Lm)$ with some substitution Φ and $H=RHS\Phi$. Let a labeled version of this rule be $f^*(L,L1^*,...,Lm^*)=>RHS^*$. It is easily verified that $E0^*$ matches the head of this rule with substitution Φ^* such that $\langle L,\alpha \rangle$ is in Φ^* and for each pair $\langle X,t \rangle$ in Φ , the pair $\langle X,t^* \rangle$ is in Φ^* where t^* is a labeled version of t . It is also easily verified that $RHS^*\Phi^*$ is a labeled version of $RHS\Phi$. QED.

Theorem 3. Derived minimality for DF*. Let P be a DF* program and $E0$ a term, where both P and $E0$ are ordinary. Let P^* and $E0^*$ be, respectively, their labeled versions such that $E0^*$ is proper. Let $E0,E1,...,Ek$ be a successful reduction in P . Then there exists a successful NA-derivation $E0^*,F1^*,...,Fp^*$ in P^* such that $p=<k$.

Proof. We can ensure that $E0^*$ is a labeled version of $E0$ which is proper, simply by choosing distinct maximal labels for function symbols of $E0$. By Lemma 12, there exists a successful reduction $E0^*,E1^*,...,Ek^*$ such that for each i , Ei^* is a labeled version of Ei . Since P^* is an LDF* program, and $E0^*$ is proper, by Theorem 2, there exists a successful NA-derivation $E0^*,F1^*,...,Fp^*$ such that $p=<k$. QED.

V. COMPILATION OF F* INTO HORN CLAUSES

V.1 INTRODUCTION

A very simple algorithm is described, which compiles F* programs into Horn clauses in such a way that when SLD-resolution interprets them, it directly simulates the behavior of select. This is accomplished by compiling each F* rule into a distinct Horn clause, and combining in that clause, information about the logic of the rule, and information about the control of select when interpreting that rule. Thus, a specialized interpreter is produced for each rule.

If the F* program satisfies restriction (g) in Section III.2, the clauses resulting from its translation can be transformed to eliminate all redundant backtracking. If the program also satisfies restriction (f), i.e. is in DF*, SLD-search trees automatically contain exactly one branch. All the time, however, only *pure* clauses are produced.

The nature of logical variables is utilized to implement the assumption necessary for minimality. This is that when a term is reduced, all copies of it are simultaneously reduced. A logical variable has the property that when one occurrence of it in a term is bound to some term, all occurrences of it are simultaneously bound to the same term. Unfortunately, use must now be made of a metalogical feature (var), and an extra logical feature (cut). This is the *only* impure aspect in the entire LOG(F) system. Consequently, SLD-resolution, augmented with these features, computes NA-reductions.

LOG(F) is defined to be a logic programming system augmented with an F* compiler, and the equality axiom $X=X$. A ready-made implementation of LOG(F) is obtained by implementing the F* compiler in Prolog and using Prolog in place of SLD-resolution. Due to its depth-first search strategy, Prolog may sometimes not be able to simplify terms, even though select would. *However, if P is in DF*, Prolog always simplifies terms whenever select does.*

In all of the following, except in Section V.6, Prolog clauses and Prolog are synonymous with Horn clauses and SLD-resolution. Only in Section V.6 do they refer to the *programming language*.

V.2 COMPILATION ALGORITHM

Let P be an F* program. The compilation of P into Prolog proceeds in two stages.

Stage 1. For each n-ary, $n \geq 0$, constructor symbol c in P, and where X_1, \dots, X_n are distinct variables, generate the clause:

`reduce(c(X1,...,Xn),c(X1,...,Xn))`

Stage 2. Let $f(L1,...,Lm) \Rightarrow RHS$ be a rule in P where f is an m -ary, $m \geq 0$, non-constructor function symbol and each of RHS and $L1,...,Lm$ is a term, possibly containing variables. For each such rule perform the following steps:

(a) Let $A1,...,Am$ be distinct Prolog variables none of which occur in the rule. If Li is a variable let Qi be $Ai=Li$. If Li is $c(X1,...,Xn)$ where c is a constructor symbol, and each Xi a variable, let Qi be `reduce(Ai,c(X1,...,Xn))`.

(b) Let Out be a Prolog variable not occurring in the rule, and different from $A1,...,Am$. Generate the predication `reduce(RHS,Out)`.

(c) Generate the clause:

`reduce(f(A1,...,Am),Out):-Q1,...,Qm,reduce(RHS,Out).`

For example the F* rules:

```
append([],X)=>X
append([U|V],W)=>[U|append(V,W)]
intfrom(N)=>[N|intfrom(s(N))].
if(true,X,Y)=>X.
if(false,X,Y)=>Y.
```

are compiled into:

```
reduce([],[]).
reduce([U|V],[U|V]).
reduce(true,true).
reduce(false,false).

reduce(append(A1,A2),Out):-reduce(A1,[],A2=X,reduce(X,Out).
reduce(append(A1,A2),Out):-
    reduce(A1,[U|V]),A2=W,reduce([U|append(V,W)],Out).
reduce(intfrom(N),Out):-reduce([N|intfrom(s(N))],Out).
reduce(if(T,X,Y),Out):-reduce(T,true),reduce(X,Out).
reduce(if(T,X,Y),Out):-reduce(T,false),reduce(Y,Out).
```

It can be seen that where `reduce(f(A1,...,Am),Out):-Q1,...,Qm,reduce(RHS,Out)` is the translation of $f(L1,...,Lm) \Rightarrow RHS$, $Q1,...,Qm$ represent the attempt to match some term

$f(t_1, \dots, t_m)$ with $f(L_1, \dots, L_m)$. If these succeed, the match succeeds with some substitution α . Now, $\text{reduce}(\text{RHS}, \text{Out})$ represents simultaneously, application of α to RHS, and recursive simplification of $\text{RHS}\alpha$. The correctness of compilation algorithm is formally proved in Section V.7.

In practice, in stage 2(a) if L_i is a variable, then A_i in $f(A_1, \dots, A_m)$ is replaced by L_i , and $A_i = L_i$ is not generated. This eliminates a procedure call, and so yields substantially faster code. However, proofs of propositions below are easier to derive without this optimization.

V.3 COMPUTING AND PRINTING NORMAL FORMS

If there is a method to compute simplified forms of terms, it can be applied repeatedly to compute normal forms of terms. *This is guaranteed by reduction-completeness for normal forms, Theorem 4, Section II.* In particular, for each m -ary constructor symbol we can add the following rule:

$$\text{nf}(E, c(X_1, \dots, X_m)) :- \text{reduce}(E, c(T_1, \dots, T_m)), \text{nf}(T_1, X_1), \dots, \text{nf}(T_m, X_m).$$

Now, to compute the normal form of a term E , we can execute $\text{nf}(E, X)$, where X is a variable. The correctness of this rule for computing normal forms can easily be proved from the arguments of Section V.7.

Clearly, computing normal forms is only sensible when they are finite. If they are not, we can at least print finite portions of them as they are generated. For example, we can print members of an infinite list as follows:

$$\text{print_list}(X) :- \text{reduce}(X, [U|V]), \text{write}(U), \text{write}(' '), \text{print_list}(V).$$

V.4 OPTIMIZING RULES SATISFYING RESTRICTION (g)

Let P be an F* program and PC its compiled version. Let $f(t_{l1}, \dots, t_{li}, \dots, t_{lm}) \Rightarrow \text{RHS}_1, \dots, f(t_{n1}, \dots, t_{ni}, \dots, t_{nm}) \Rightarrow \text{RHS}_n$ be the n rules defining f in P , and C_1, \dots, C_n be, respectively, their compiled versions. Let the rules satisfy restriction (g), Section III.2. Then, if t_{li} is a variable, the i th literal in bodies of C_1, \dots, C_n will be, respectively, $A_i = t_{li}, \dots, A_i = t_{ni}$, for some variable A_i . Otherwise, the i th literals in C_1, \dots, C_n would be, respectively, $\text{reduce}(A_i, t_{li}), \dots, \text{reduce}(A_i, t_{ni})$.

If t_{li} is not a variable, the query $\text{reduce}(f(a_1, \dots, a_i, \dots, a_m), Z)$ may, due to backtracking, cause evaluation of each of $\text{reduce}(a_i, t_{li}), \dots, \text{reduce}(a_i, t_{ni})$. We can ensure that reduce is called just once for a_i by taking advantage of the fact that all reduce clauses have the

same form. That is, we can collapse them all into the single clause:

$$\text{reduce}(f(A_1, \dots, A_m), Z) :- R_1, \dots, R_m, f(X_1, \dots, X_m) \Rightarrow \text{RHS}, \text{reduce}(\text{RHS}, Z).$$

where X_1, \dots, X_m are distinct variables not occurring in any of the clauses, and if t_{1i} is a variable, R_i is $A_i = X_i$, otherwise R_i is $\text{reduce}(A_i, X_i)$. Now reduce would be called just once for a_i . Of course, the \Rightarrow rules now need to be included with the reduce clauses. Thus Prolog execution can be considerably speeded up.

Furthermore, if P is a DF* program then $f(X_1, \dots, X_m) \Rightarrow \text{RHS}$ will succeed at most once. Hence, for any ground terms t_1, \dots, t_m , and variable Z , the search tree rooted at $\text{reduce}(f(t_1, \dots, t_m), Z)$ will contain exactly one branch. *Thus, the reduce clauses would form a deterministic logic program.* For example, consider the DF* program:

```
append([], X) => X.
append([U|V], W) => [U|append(V, W)].
```

Its compiled version, excluding rules for constructor symbols, is:

```
reduce(append(A1, A2), Z) :- reduce(A1, []), A2 = X, reduce(X, Z).
reduce(append(A1, A2), Z) :-
    reduce(A1, [U|V]), A2 = W, reduce([U|append(V, W)], Out).
```

These two rules can be collapsed into a single one:

```
reduce(append(A1, A2), Z) :-
    reduce(A1, X1), A2 = X2, append(X1, X2) => RHS, reduce(RHS, Z).
```

Now, given the query $\text{reduce}(\text{append}([1], [2]), Z)$, an attempt would be made to simplify $[1]$ just once, and not twice, as with the original pair of reduce clauses. Also, since the append rules are in DF*, the SLD-search tree rooted at $\text{reduce}(\text{append}([1], [2]), Z)$ contains exactly one branch.

V.5 COMPUTING FUNCTIONS EAGERLY IN F*

If a function is defined in F*, it is computed lazily. Often it is very desirable that some functions, such as arithmetic functions, be computed eagerly. We show one way to accomplish this.

A lazy function symbol is one which is defined in F*. An eager function symbol is one which is defined in Prolog. Only right hand sides of F* rules can contain calls to eager

functions. Let E be a subterm, possibly containing variables, of the right hand side of an F* rule. Let the outermost function symbol of E be eager. Then E must not contain any lazy function symbol. For example, where length is eager, and append is lazy, the term `length(append([], [1]))` must not appear in any F* rule.

Now, let $LHS \Rightarrow RHS$ be an F* rule, f an eager function, and $f(t_1, \dots, t_n)$ a subterm, possibly containing variables, of RHS. Let f be defined by an $n+1$ ary predicate symbol $p(A_1, \dots, A_n, A)$, such that A_1, \dots, A_n are input positions and A the output position. Let RHS_1 be the result of replacing $f(t_1, \dots, t_n)$ in RHS by X , where X is a variable not occurring in $LHS \Rightarrow RHS$. Generate the condition $p(t_1, \dots, t_n, X)$, and add it to the conditions generated in Stage 2 (a) of Section V.2. Of course, if t_1, \dots, t_n themselves involve calls to eager functions, they must be treated similarly. For example, let `multiple` be an eager function defined in Prolog as follows:

```
multiple(A,B,true):-0 is A mod B.
multiple(A,B,false):-not(0 is A mod B).
```

Now the rule:

```
filter(A,[U|V])=>if(multiple(U,A),filter(A,V),[U|filter(A,V)]).
```

is compiled into:

```
reduce(filter(A,X),Z):-
    reduce(X,[U|V]),
    multiple(U,A,T),
    reduce(if(T,filter(A,V),[U|filter(A,V)]),Z).
```

However, some care still needs to be exercised. For example, where `zerop` and `/` are eager functions, defined in Prolog by, respectively, `zerop` and `div`, the rule:

```
f(X)=>if(zerop(X),[X],[1/X]).
```

will be compiled into:

```
reduce(f(X),Z):-zerop(X,T),div(1,X,A),reduce(if(T,[X],[A]),Z).
```

Now, if X is 0, the call to `div` will cause an unintended division by 0. But one can rewrite the above rule as:

```
f(X)=>if(zerop(X),[X],h(X)).
```

$$h(X) \Rightarrow [1/X].$$

V.6 COMPILING LDF* PROGRAMS

We now show how to represent labeled terms in Prolog, and compile LDF* programs into Prolog in such a way that NA-steps can be performed efficiently. The main idea is that labels can be represented by logical variables. These have the property that if one occurrence of a variable in term E is bound to term F , all occurrences of the variable in E are simultaneously bound to F .

Let E be a proper term and let E reduce to F in an NA-step. Then there is a subterm G of E such that $G \Rightarrow H$, and F is obtained by replacing all occurrences of G in E by H . Note that each of G, H, F is proper. Let E contain the labels $\alpha_1, \dots, \alpha_n$. Let V_1, \dots, V_n be distinct variables and E^* the result of replacing for each i , all occurrences of α_i in E by V_i . Then E^* is a Prolog representation of E . Similarly, let G^*, H^*, F^* be Prolog representations of G, H, F respectively such that H^* and E^* do not have any variables in common. Then $G^* = f(V, t_1, \dots, t_m)$ where V is a variable. If we now bind V to H^* , all occurrences of V in E^* are bound to H^* . Let the result be $F1^*$.

Now, before attempting to match a term with a non-variable term, we take the precaution of checking whether its label is already bound to some term. If so, we attempt to match this term with the non-variable term. Otherwise, we proceed as usual. Thus, after V has been bound to H^* , if another occurrence of G^* is to be matched with some term, we attempt to match H^* with it.

At a later stage it is possible that the label of H^* itself be bound to a term. Thus, before matching a term, it may be necessary to "dereference" its label a number of times. It is not unreasonable to assume that the cost of dereferencing is small compared to that of reduction. Thus, we can work with $F1^*$ instead of F^* , so replacement of all occurrences of a term is implemented efficiently. Moreover, F^* can be obtained from $F1^*$ by dereferencing. The algorithm for compiling LDF* programs can be found in [Narain 1988]. In practice, DF* programs can be compiled directly into reduce clauses with labels, without first transforming them into LDF* programs. The appropriate algorithm can easily be worked out.

It will be recognized that our scheme for implementing NA-derivations is exactly the graph-reduction scheme with indirection nodes described in [Turner 1979] and [O'Donnell 1982]. *However, in our case it is possible to ensure that the length of the dereferencing chain is exactly one.* Details can be obtained in [Narain 1988]. For example, where nil is a zero-ary constructor symbol, let P be the following DF* program:

```

merge(nil,nil)=>nil.
double(X)=>merge(X,X).
h=>d.

```

This is compiled into:

- (1) $\text{reduce}(\text{merge}(V,A1,A2),Z):-\text{not var}(V),\text{reduce}(V,Z),!.$
- (2) $\text{reduce}(\text{double}(V,A1),Z):-\text{not var}(V),\text{reduce}(V,Z),!.$
- (3) $\text{reduce}(h(V),Z):-\text{not var}(V),\text{reduce}(V,Z),!.$
- (4) $\text{reduce}(\text{nil}(N),\text{nil}(N)).$
- (5) $\text{reduce}(\text{merge}(Z,A1,A2),Z):-$
 $\quad \text{reduce}(A1,\text{nil}(N1)),\text{reduce}(A2,\text{nil}(N2)),\text{reduce}(\text{nil}(N3),Z).$
- (6) $\text{reduce}(\text{double}(Z,A1),Z):-\text{reduce}(\text{merge}(N,A1,A1),Z).$
- (7) $\text{reduce}(h(Z),Z):-\text{reduce}(d(D),Z).$

Consider the query $\text{reduce}(\text{double}(A,h(B)),Z)$, which has as descendant $\text{reduce}(\text{merge}(N,h(B),h(B)),Z)$. Suppose the first call, $\text{reduce}(h(B),\text{nil}(N1))$, in (5) succeeds, but only after a long and complicated deduction. Then B is bound to $\text{nil}(N1)$. Now, due to (3), the second call, $\text{reduce}(h(B),\text{nil}(N2))$, in (5) will terminate in just three inference steps. The cut (!) will prevent (7) from being tried all over again. Also note that of these three inference steps, only one is a dereferencing step. The number of these steps is constant, regardless of the function definitions.

V.7 CORRECTNESS OF F* COMPILATION ALGORITHM

Lemma 1. Let P be an F* program. If:

- (1) $E0=f(t1,...,ti,...,tm)$, and
- (2) $Ek=f(s1,...,si,...,sm)$, and
- (3) si is simplified, and
- (4) $E0,...,Ek, k \geq 0$, is an N-reduction such that for no $i, Ei \Rightarrow Ei+1$.

Then there is a successful N-reduction $ti,...,si$ of length less than or equal to the length k of $E0,E1,...,Ek$.

Proof: By Lemma 8, Section III. QED.

Lemma 2. Let P be an F* program, and PC its compiled version. Let A be a ground term and B a term, possibly containing variables, such that $\text{reduce}(A,B)$ succeeds, in the sense of SLD-resolution, with answer substitution σ . Then $B\sigma$ is ground.

Proof: By induction on length n of successful SLD-derivation $\text{reduce}(A,B), G_1, \dots, G_n = \square$.
QED.

Lemma 3. Let P be an F* program and PC its compiled version. Let A and B be ground terms such that $\text{reduce}(A,B)$ succeeds. Let D be a term, possibly containing variables, such that for some substitution α , $D\alpha = B$. Then $\text{reduce}(A,D)$ succeeds with answer substitution α .

Proof: By induction on length n of successful SLD-derivation starting at $\text{reduce}(A,B)$.
QED.

Lemma 4. Let P be an F* program. Let PC be the compiled version of P . Let E_0, \dots, E_n be a successful N-reduction. Then $\text{reduce}(E_0, E_n)$ succeeds in the presence of PC .

Plan of Proof: By induction on length of successful N-reduction E_0, \dots, E_n . We show that there is some E_j , $j > 0$, in E_0, \dots, E_n such that an SLD-derivation of $\text{reduce}(E_0, E_n)$ contains the goal $\text{reduce}(E_j, E_n)$. Since E_j, \dots, E_n is also a successful N-reduction, by induction hypothesis, $\text{reduce}(E_j, E_n)$ succeeds. Hence $\text{reduce}(E_0, E_n)$ succeeds.

Proof: By induction on length n of successful reduction E_0, \dots, E_n . If $n=0$ then E_0 is already simplified. In particular, $E_0 = c(t_1, \dots, t_m)$ where c is an m -ary constructor symbol, $m \geq 0$, and t_1, \dots, t_m are terms. There is a clause in PC $\text{reduce}(c(X_1, \dots, X_m), c(X_1, \dots, X_m))$ where each X_i is a variable. Clearly $\text{reduce}(E_0, E_0)$ succeeds.

Let $n > 0$ and $E_0 = f(t_1, \dots, t_m)$, f not a constructor symbol, each t_i a term and $m \geq 0$. Assume theorem holds for all successful reductions of length less than n .

Since E_0 is not simplified, the N-reduction is of the form $E_0, \dots, E_{k-1}, E_k, \dots, E_n$, $0 < k \leq n$, such that $E_{k-1} \Rightarrow E_k$, but for no i , $0 \leq i < k-1$, $E_i \Rightarrow E_{i+1}$. Hence, $E_{k-1} = f(s_1, \dots, s_m)$ for some terms s_1, \dots, s_m . Since $E_{k-1} \Rightarrow E_k$, there is some rule $f(L_1, \dots, L_m) \Rightarrow \text{RHS}$ such that E_{k-1} matches $f(L_1, \dots, L_m)$ with some substitution σ and $E_k = \text{RHS}\sigma$. Since L_1, \dots, L_m do not share any variables, σ is the union of $\sigma_1, \dots, \sigma_m$ such that for each L_i in L_1, \dots, L_m , s_i matches L_i with substitution σ_i .

For each i , if L_i is not a variable, then since s_i matches L_i , s_i is in simplified form. For such i , there is, by Lemma 1, a successful N-reduction t_i, \dots, s_i of length less than or equal to $k-1$.

The rule $f(L_1, \dots, L_m) \Rightarrow \text{RHS}$ is compiled into the Horn clause

$$\text{reduce}(f(X_1, \dots, X_m), Z) :- Q \cup \{\text{reduce}(\text{RHS}, Z)\}$$

in accordance with the compilation rules stated above. This clause is contained in PC.

Consider the query $\text{reduce}(E_0, En)$, i.e. $\text{reduce}(f(t_1, \dots, t_m), En)$. It unifies with $\text{reduce}(f(X_1, \dots, X_m), En)$ with m.g.u. $\tau = \{ \langle X_1, t_1 \rangle, \dots, \langle X_m, t_m \rangle, \langle Z, En \rangle \}$ and its immediate descendant is $(Q \cup \{ \text{reduce}(RHS, Z) \})\tau$. Since RHS does not contain any of the X_i , this is $Q\tau \cup \{ \text{reduce}(RHS, En) \}$.

Let Q_1, \dots, Q_m be the members of Q . Consider some Q_i . If Q_i is $X_i = Li$, then $Q_i\tau = (t_i = Li)$ which succeeds with answer substitution $\{ \langle Li, t_i \rangle \}$. Of course, t_i matches Li , so $\{ \langle Li, t_i \rangle \} = \sigma_i$.

Otherwise, $Q_i = \text{reduce}(X_i, Li)$, so $Q_i\tau = \text{reduce}(t_i, Li)$. Since there is a successful N-reduction t_i, \dots, si of length less than or equal to $k-1$, by induction hypothesis, $\text{reduce}(t_i, si)$ succeeds. Since $Li\sigma_i = si$, by Lemma 3, $\text{reduce}(t_i, Li)$ also succeeds with answer substitution σ_i .

By repeating the same argument for each Q_i , we see that an SLD-derivation starting at $\text{reduce}(E_0, En)$ contains $\text{reduce}(RHS\sigma_1, \dots, \sigma_m, En)$ as a member. Since σ is the union of $\sigma_1, \dots, \sigma_m$ and no variable is defined in more than one σ_i in $\sigma_1, \dots, \sigma_m$, $RHS\sigma_1, \dots, \sigma_m = RHS\sigma$. But $RHS\sigma = E_k$. Hence the SLD-derivation starting at $\text{reduce}(E_0, En)$ contains $\text{reduce}(E_k, En)$. Since the length of the successful reduction E_k, \dots, En is less than n , by induction hypothesis, $\text{reduce}(E_k, En)$ succeeds. Thus, the query $\text{reduce}(E_0, En)$ succeeds. QED.

Lemma 5. Let P be an F^* program. Let PC be the compiled version of P . Let E_0 and En be terms such that $\text{reduce}(E_0, En)$ succeeds in the presence of PC. Then there is a successful N-reduction E_0, \dots, En .

Plan of Proof: By induction on length of successful SLD-derivation $\text{reduce}(E_0, En), \dots, \square$. We show that there is some goal $\text{reduce}(E_j, En)$, $j > 0$, in this derivation such that there is an N-reduction E_0, \dots, E_j . Since $\text{reduce}(E_j, En)$ succeeds, by induction hypothesis, there is a successful N-reduction E_j, \dots, En . So there is a successful N-reduction $E_0, \dots, E_j, \dots, En$.

Proof: By induction on length n of successful SLD-derivation starting at $\text{reduce}(E_0, En)$. If $n=1$ then there is a clause $\text{reduce}(c(X_1, \dots, X_m), c(X_1, \dots, X_m))$ in PC such that $\text{reduce}(E_0, En)$ unifies with the head of this clause. Clearly, then, $E_0 = En$, En is simplified and the required N-reduction is simply E_0 .

Let $n > 0$. Assume lemma for all successful derivations of length less than n . Assume $E_0 = f(t_1, \dots, t_m)$ for some non-constructor function symbol f and terms t_1, \dots, t_m . Since $\text{reduce}(E_0, En)$ succeeds there is a clause in PC:

$$\text{reduce}(f(X1, \dots, X_m), Z) :- Q \cup \{\text{reduce}(\text{RHS}, Z)\}$$

such that it is the compilation of a rule $f(L1, \dots, L_m) \Rightarrow \text{RHS}$ in P. Moreover, $\text{reduce}(f(t1, \dots, t_m), \text{En})$ unifies with the head of the above clause with m.g.u. $\tau = \{ \langle X1, t1 \rangle, \dots, \langle X_m, t_m \rangle, \langle Z, \text{En} \rangle \}$ and $Q\tau \cup \{\text{reduce}(\text{RHS}, Z)\}\tau$ has a successful derivation of length $n-1$. Also, $\text{RHS}\tau = \text{RHS}$ and $Z\tau = \text{En}$.

If Q is empty, $m=0$. So, by restriction (e) RHS is ground. By induction hypothesis there is a successful N-reduction $\text{RHS}, \dots, \text{En}$. $E0$ matches $f(L1, \dots, L_m)$ and so $E0 \Rightarrow \text{RHS}$. Hence $E0, \text{RHS}, \dots, \text{En}$ is a successful N-reduction.

Suppose Q is non-empty. Let $Q1, \dots, Q_m$ be the members of Q. Consider Qi . If $Qi = (Xi = Li)$ then ti unifies with Li with substitution $\sigma_i = \{ \langle Li, ti \rangle \}$. Construct the singleton sequence $f(t1, \dots, ti, \dots, t_m)$. This sequence is an N-reduction.

If $Qi = \text{reduce}(Xi, Li)$ then $Li = c(U1, \dots, Uk)$ for some constructor symbol c and variables $U1, \dots, Uk$. Also $Qi\tau = \text{reduce}(ti, Li)$. Clearly, $\text{reduce}(ti, Li)$ succeeds. Let the answer substitution be σ_i . By Lemma 2, $Li\sigma_i$ is ground. Then $\text{reduce}(ti, Li\sigma_i)$ also succeeds. The successful derivation of $\text{reduce}(ti, Li\sigma_i)$ is the same as that of $\text{reduce}(ti, Li)$ with Li replaced by $Li\sigma_i$. So, the length of this derivation is also less than n . By induction hypothesis, there is a successful N-reduction $ti, \dots, Li\sigma_i$. By Lemma 4 of Section II, the sequence $f(t1, \dots, ti, \dots, t_m), \dots, f(t1, \dots, Li\sigma_i, \dots, t_m)$ is an N-reduction.

Hence we obtain the N-reductions $f(t1, \dots, t_m), \dots, f(L1\sigma1, \dots, t_m)$ and $f(L1\sigma1, t2, \dots, t_m), \dots, f(L1\sigma1, L2\sigma2, \dots, t_m)$ and .. $f(L1\sigma1, L2\sigma2, \dots, t_m), \dots, f(L1\sigma1, L2\sigma2, \dots, L_m\sigma_m)$. The concatenation of these reductions is itself an N-reduction. Since $L1, \dots, L_m$ do not share variables, $f(L1\sigma1, \dots, L_m\sigma_m)$ matches $f(L1, \dots, L_m)$ with a substitution which is the union of $\sigma1, \dots, \sigma_m$. Let σ be this union. Hence $f(L1\sigma1, \dots, L_m\sigma_m) \Rightarrow \text{RHS}\sigma$. Since all the variables of RHS are in $L1, \dots, L_m$ and for each σ_i , $Li\sigma_i$ is ground, $\text{RHS}\sigma$ is ground.

The predication $\text{reduce}(\text{RHS}\sigma, \text{En})$ succeeds and the length of the associated successful derivation is less than n . By induction hypothesis, there is a successful N-reduction $\text{RHS}\sigma, \dots, \text{En}$. Hence there is a successful N-reduction $f(t1, \dots, t_n), \dots, f(L1\sigma1, \dots, L_m\sigma_m), \text{RHS}\sigma, \dots, \text{En}$. QED.

Theorem 1. The correctness of the compilation of F*. Let P be an F* program and PC be its compilation. Let $E0$ and En be ground terms. Then there is a successful N-reduction beginning with $E0$ and ending with En iff $PC \vdash \text{reduce}(E0, En)$.

Proof: Lemmas 4 and 5 state, respectively, the if and only if parts of the theorem. QED.

VI. PROGRAMMING IN LOG(F)

VI.1 INTRODUCTION

This section describes seven examples of programming in LOG(F). The first illustrates non-determinism of LOG(F), and usefulness of lazy evaluation even when manipulating finite data structures. The second shows how useful cases of the rule of substitution of equals for equals can be implemented. The third obtains a new proof of confluence of combinatory logic. The fourth shows how a pair of communicating processes can be simulated. The fifth illustrates the power of NA-derivations, and manipulation of infinite numerical structures. The sixth illustrates manipulation of infinite graphical structures. The seventh compares LOG(F) with the system of Tamaki [1984]. In each case, clauses listed are those obtained after performing optimizations discussed in previous sections. Finally, Section VI.9 compares performance of LOG(F) with that of Prolog.

VI.2 NON-DETERMINISM IN F*

As discussed in Section I, permutations of lists can be computed by the following F* program:

```
perm([])=>[].
perm([U|V])=>insert(U,perm(V)).
insert(U,X)=>[U|X].
insert(U,[A|B])=>[A|insert(U,B)].
```

This is compiled, and optimized into:

```
reduce([],[]).
reduce([A|B],[A|B]).

reduce(insert(A,B),[A|B]).
reduce(insert(A,B),[C|insert(A,D)]):-reduce(B,[C|D]).
reduce(perm(A),B):-reduce(A,C),perm(C)=>D,reduce(D,B).

perm([])=>[].
perm([A|B])=>insert(A,perm(B)).
```

Note that some => rules survive in the compiled version. This is due to the method, discussed in Section V.4, of compiling F* programs satisfying restriction (g). If we now type `reduce(perm([1,2,3]),Z)`, we obtain `Z=[1|perm([2,3])]`, `Z=[2|insert(1,perm([3]))]`, `Z=[3|insert(1,insert(2,perm([1])))]`. However, if we define:

```
make_list(X,[]):-reduce(X,[]).
make_list(X,[U|V]):-reduce(X,[U|B]),make_list(B,V).
```

and then type `make_list(perm([1,2,3]),Z)`, we obtain `Z=[1,2,3],...,Z=[3,2,1]`.

The above program can be used to implement a very efficient solution to the N-queens problem which is to place N queens on an NxN chess board so that no two queens attack each other. It is easily seen that each queen must be in a distinct row and column, so that candidates for solutions can be represented by permutations of the list `[1,2,...,N]`. The position of the *i*th queen in a permutation *p* is `[i,q]` where *q* is the *i*th element of *p*. The problem now reduces to generating all permutations of `[1,2,...,N]` and testing whether they are safe, or represent a solution.

Lazy evaluation guarantees that permutations are tested as soon as they are generated. If it is determined that `[A1,...,Am]`, $m \leq N$ is unsafe then no permutation with `[A1,...,Am]` as initial segment is generated. This yields a drastic pruning of the search space. The program is:

```
if(true,X,Y)=>X.
if(false,X,Y)=>Y.
queens(X)=>safe(perm(X)).
safe([])=>[].
safe([U|V])=>[U|safe(nodiagonal(U,V,1))].
nodiagonal(U,[],N)=>[].
nodiagonal(U,[A|B],N)=>if(noattack(U,A,N),[A|nodiagonal(U,B,N+1)],none).
noattack(U,A,N)=>neg(equal(abs(U-A),N)).
```

This is compiled into:

```
reduce([],[]).
reduce([U|V],[U|V]).
reduce(true,true).
reduce(false,false).

reduce(queens(A),B):-queens(A)=>C,reduce(C,B).
reduce(safe(A),B):-reduce(A,C),safe(C)=>D,reduce(D,B).
reduce(if(A,B,C),D):-reduce(A,E),if(E,B,C)=>F,reduce(F,D).
reduce(noattack(A,B,C),D):-noattack(A,B,C)=>E,reduce(E,D).
reduce(nodiagonal(A,B,C),D):-reduce(B,E),nodiagonal(A,E,C)=>F,reduce(F,D).

queens(A)=>safe(perm(A)).
```

```

safe([])=>[].
safe([A|B])=>[A|safe(nodiagonal(A,B,1))].
if(true,A,B)=>A.
if(false,A,B)=>B.
nodiagonal(A,[],B)=>[].
nodiagonal(A,[B|C],D)=>
    if(noattack(A,B,D),[B|nodiagonal(A,C,E)],none):-E is D+1.
noattack(A,B,C)=>D:-E is A-B,abs(E,F),equal(F,C,G),neg(G,D).

```

The eager functions are defined in Prolog:

```

abs(X,X):-X>=0.
abs(X,Y):-X<0,Y is -X.
neg(true,false).
neg(false,true).
equal(A,A,true).
equal(A,B,false):-not A=B.
less_than(U,A,true):-U<A.
less_than(U,A,false):-U>=A.

```

If we now type `make_list(queens([1,2,3,4]),Z)`, we obtain `Z=[2,4,1,3]` and `Z=[3,1,4,2]`.

VI.3 IMPLEMENTING SUBSTITUTION OF EQUALS FOR EQUALS

If a DF* program is interpreted as an equality theory, reduce clauses can be thought of as implementing an equality theory in Prolog with the restriction that it be used only for simplification of terms. Now, given a clause of the form `p(c(X1,...,Xm)):-Body`, where `c` is a constructor symbol, we can add another clause stating a rule of substitution of equals:

```
p(X):-reduce(X,c(X1,...,Xm)),p(c(X1,...,Xm)).
```

Now, even when a term `E` is not of the form `c(X1,...,Xm)`, `p` can still be inferred for `E`, provided `E` is reducible to a term of the form `c(X1,...,Xm)`. For example, from:

```

married(X):-spouse(X,Y).
spouse(scott,a).

```

one can infer `married(scott)`. One can now add the clause:

```
married(X):-reduce(X,Y),married(Y).
```

An equality theory is:

```
author(waverley)=>author(ivanhoe).
author(ivanhoe)=>scott.
```

The reduce clauses for the last two => rules are:

```
reduce(scott,scott).
reduce(ivanhoe,ivanhoe).
reduce(waverley,waverley).

reduce(author(X),Z):-reduce(X,waverley),reduce(author(ivanhoe),Z).
reduce(author(X),Z):-reduce(X,ivanhoe),reduce(scott,Z).
```

Here scott, waverley, and ivanhoe are constructor symbols. Now one can infer, in Prolog, married(author(waverley)), i.e. the result of substituting author(waverley) for scott in married(scott).

VI.4 COMBINATORY LOGIC

A new proof is obtained of the theorem that the SKI calculus is confluent. Following the ideas of Ait-Kaci & Nasr [1986], SKI reduction rules can be expressed as a DF* program:

```
apply(k,X)=>k1(X).
apply(k1(X),Y)=>X.
apply(s,F)=>s1(F).
apply(s1(F),G)=>s2(F,G).
apply(s2(F,G),X)=>apply(apply(F,X),apply(G,X)).
```

Here k,s,k1,s1,s2 are constructor symbols, and apply a non-constructor symbol. From confluence of DF*, it follows that the SKI calculus is also confluent. These rules are translated into the following reduce clauses:

```
reduce(s,s).
reduce(k,k).
reduce(k1(X),k1(X)).
reduce(s1(X),s1(X)).
reduce(s2(X,Y),s2(X,Y)).

reduce(apply(A,B),Z):-reduce(A,k),reduce(k1(B),Z).
```

```

reduce(apply(A,B),Z):-reduce(A,k1(D)),reduce(D,Z).
reduce(apply(A,B),Z):-reduce(A,s),reduce(s1(B),Z).
reduce(apply(A,B),Z):-reduce(A,s1(C)),reduce(s2(C,B),Z).
reduce(apply(A,B),Z):-
    reduce(A,s2(D,E)),reduce(apply(apply(D,B),apply(E,B)),Z).

```

These clauses can be used to contemplate higher-order programming in LOG(F).

VI.5 TWO WAY COMMUNICATION

This example models communication between two users, each of whom types a stream of tokens on his screen. Each token is of the form [A] or [send,M] in which case M appears on both screens. The communication is modeled by:

```

extract_messages([[A]|X])=>extract_messages(X).
extract_messages([[send,M]|X])=>[M|extract_messages(X)].

screen1=>fair_merge(key1,extract_messages(key2)).
screen2=>fair_merge(key2,extract_messages(key1)).

```

Here send, [] and | are constructor symbols. We assume there exists a function fair_merge which takes as input two streams and interleaves their tokens into an output stream. If two tokens appear in some order in an input, then they appear in the same order in the output. Finally, fair_merge consumes each input at the rate at which it is produced.

Note that the second extract_messages rule has a left hand side of depth greater than two, so, strictly speaking, it is not an F* rule. However, it can be expressed in F* as follows:

```

extract_messages([A|X])=>g(A,X).
g([U|V],X)=>h(U,V,X).
h(send,V,X)=>[V|extract_messages(X)].

```

Here g and h are auxiliary function symbols. Assuming that key1 and key2 are streams of tokens typed by, respectively, the first and second user, the term screen1 will reduce to the stream of tokens appearing on the first user's screen. Similarly for screen2. The reduce clauses are:

```

reduce([],[]).
reduce([U|V],[U|V]).
reduce(send,send).

```

```

reduce(extract_messages(A),B):-
    reduce(A,[C|D]),reduce(C,[E]),reduce(extract_messages(D),B).
reduce(extract_messages(A),[B|extract_messages(C)]):-
    reduce(A,[D|C]), reduce(D,[E|F]), reduce(E,send), reduce(F,[B]).
reduce(screen1,A):-reduce(fair_merge(key1,extract_messages(key2)),A).
reduce(screen2,A):-reduce(fair_merge(key2,extract_messages(key1)),A).

```

VI.6 HAMMING'S PROBLEM

The problem, described in [Dijkstra 1976], is to generate, in increasing order, all those numbers which are divisible by no primes other than 2,3 or 5. Dijkstra states that an equivalent problem is to generate the sequence of numbers, in ascending order, defined by the following axioms:

- (a) 1 is in the sequence
- (b) If x is in the sequence, then so are $2*x$, $3*x$ and $5*x$.
- (c) The sequence contains no values except those on account of (a) and (b).

These axioms can be expressed by the following DF* program:

```

hamming=>hamming_aux([1|hamming]).
hamming_aux(X)=>
    merge(times_list(2,X),merge(times_list(3,X),times_list(5,X))).

merge([U|V],[A|B])=>if(U<A,[U|merge(V,[A|B])],merge_aux(U,V,A,B)).
merge_aux(U,V,A,B)=>if(equal(U,A),[U|merge(V,B)],[A|merge([U|V],B)]).

times_list(N,[])=>[].
times_list(N,[U|V])=>[U*N|times_list(N,V)].

```

Function `times_list` multiplies each element of its input list by a fixed number. Function `merge` takes two lists in ascending order and merges their elements in increasing order. Functions `hamming` and `hamming_aux` are implementations of axioms (a),(b),(c).

This program illustrates the power of NA-derivations. The definition of `hamming_aux` contains three occurrences of X on the right hand side. If care is taken that whenever the term at one occurrence of X is reduced, terms at the other two occurrences of X are also reduced, the list `hamming` is produced with little overhead. If not, then the overhead increases exponentially. This can be felt by comparing the speed with which elements of `hamming` are printed on the screen in the two cases. The labeled version of this program is compiled into the following reduce clauses:


```

reduce(hamming(A),B):-not var(A),B=A,!.
reduce(hamming_aux(A,B),C):-not var(A),C=A,!.
reduce(merge(A,B,C),D):-not var(A),D=A,!.
reduce(merge_aux(A,B,C,D,E),F):-not var(A),F=A,!.
reduce(times_list(A,B,C),D):-not var(A),D=A,!.
reduce(if(A,B,C,D),E):-not var(A),E=A,!.

reduce([],[]).
reduce([A|B],[A|B]).
reduce(true,true).
reduce(false,false).

reduce(hamming(A),A):-hamming(B)=>C,reduce(C,A).
reduce(hamming_aux(A,B),A):-hamming_aux(C,B)=>D,reduce(D,A).
reduce(merge(A,B,C),A):-
    reduce(B,D),reduce(C,E),merge(F,D,E)=>G,reduce(G,A).
reduce(merge_aux(A,B,C,D,E),A):-merge_aux(F,B,C,D,E)=>G,reduce(G,A).
reduce(times_list(A,B,C),A):-reduce(C,D),times_list(E,B,D)=>F,reduce(F,A).
reduce(if(A,B,C,D),A):-reduce(B,E),if(F,E,C,D)=>G,reduce(G,A).

hamming(A)=>[1|hamming_aux(B,hamming(C))].
hamming_aux(A,B)=>
    merge(C,times_list(D,2,B),
        merge(E,times_list(F,3,B),times_list(G,5,B))).
merge(A,[B|C],[D|E])=>
    if(F,G,[B|merge(H,C,[D|E])],merge_aux(I,B,C,D,E)):-less_than(B,D,G).
merge_aux(A,B,C,D,E)=>
    if(F,G,[B|merge(H,C,E)],[D|merge(I,[B|C],E)]):-equal(B,D,G).
times_list(A,B,[])=>[].
times_list(A,B,[C|D])=>[E|times_list(F,B,D)]:-E is C*B.
if(A,true,B,C)=>B.
if(A,false,B,C)=>C.

```

Definitions of the eager functions `less_than` and `equal` are as in Section VI.2. If we now type `print_list(hamming(_))`, we obtain 1,2,3,4,5,6,8,9,10,12,...

VI.7 INFINITE GRAPHICAL STRUCTURES.

Henderson [1982] has shown how to use functional programming for defining and manipulating graphical structures. In particular, he shows how to construct Square Limit, an Escher woodcut. We use Henderson's building blocks to tile the x-y plane in an

interesting way. A picture is represented by a list of vectors, each of the form $v(A,B)--v(X,Y)$, where A,B,X,Y are real numbers. Transformations on pictures, such as composition, translation, scaling, or rotation (about the origin) are defined as follows:

```
union([],X)=>X.
union([FX|RX],Y)=>[FX|union(Y,RX)].

rotate([],_)=>[].
rotate([v(X,Y)--v(A,B)|L],Theta)=>
[v(X*cos(Theta)-Y*sin(Theta),X*sin(Theta)+Y*cos(Theta))--
v(A*cos(Theta)-B*sin(Theta),A*sin(Theta)+B*cos(Theta))|rotate(L,Theta)].

translate([],_,_)=>[].
translate([v(X,Y)--v(A,B)|L],Dx,Dy)=>
[v(X+Dx,Y+Dy)--v(A+Dx,B+Dy)|translate(L,Dx,Dy)].
scale([],_,_)=>[].
scale([v(X,Y)--v(A,B)|L],Kx,Ky)=>
[v(X*Kx,Y*Ky)--v(A*Kx,B*Ky)|scale(L,Kx,Ky)].
```

The basic pictures are p,q,r,s , drawn in a 36×36 grid, and shown in order in the top row in Figure 1. (The vectors can be found, not unfortunately, in Henderson's paper, but in [Robinson & Green 1987]). These are combined by quartet into t , shown in the second row. The third and fourth rows show, respectively, $\text{block1}(t)$ and $\text{block2}(t)$, the two basic 144×72 rectangles. $\text{row}(\text{Block},0)$ repeats Block , infinitely often, at intervals of 144 units, in the x and $-x$ directions. $\text{alt_rows}(\text{Row},0)$ repeats a row infinitely often, at intervals of 144 units, in the y and $-y$ directions. $\text{mosaic}(\text{Block1},\text{Block2})$ computes rows of Block1 and Block2 , alternates these, and then composes these to tile the x - y plane, Figure 2. The program is:

```
rot_pos(X)=>translate(rotate(translate(X,-72,-72),-1.57),72,0).
rot_neg(X)=>translate(rotate(translate(X,-72,-72),1.57),0,72).

block1(X)=>union(X,translate(rot_neg(X),72,0)).
block2(X)=>
union(rot_pos(X),
  translate(rotate(translate(rot_pos(X),-72,0),-1.57),72,0)).

row(Block,N)=>union(translate(Block,144*N,0),
  union(translate(Block,-144*N,0),row(Block,N+1))).

alt_rows(Row,N)=>union(translate(Row,0,144*N),
```

```
union(translate(Row,0,-144*N),alt_rows(Row,N+1))).
```

```
mosaic(Block1,Block2)=>union(alt_rows(row(Block1,0),0),
                             translate(alt_rows(row(Block2,0),0),0,72)).
```

```
beside(A,B)=>union(A,translate(B,36,0)).
```

```
above(A,B)=>union(A,translate(B,0,36)).
```

```
quartet(P1,P2,P3,P4)=>above(beside(P3,P4),beside(P1,P2)).
```

```
t=>quartet(p,q,r,s).
```

```
p=>[..].
```

```
q=>[..].
```

```
r=>[..].
```

```
s=>[..].
```

Note that mosaic computes an infinite row, an infinite number of times. *However, reduction-completeness of DF* precludes infinite runaway.* Vectors are displayed as they are generated. The above program is compiled into:

```
reduce(rotate(A,B),[]) :- reduce(A,[]).
```

```
reduce(rotate(A,B),[v(C,D)--v(E,F)|rotate(G,B)]) :-
```

```
    reduce(A,[H|G]), reduce(H,I--J), reduce(I,v(K,L)),
```

```
    reduce(J,v(M,N)), cos(B,O), P is K*O, sin(B,Q),
```

```
    R is L*Q, C is P-R, sin(B,S), T is K*S,
```

```
    cos(B,U), V is L*U, D is T+V, cos(B,W),
```

```
    X is M*W, sin(B,Y), Z is N*Y, E is X-Z,
```

```
    sin(B,A1), B1 is M*A1, cos(B,C1), D1 is N*C1, F is B1+D1.
```

```
reduce(translate(A,B,C),[]) :- reduce(A,[]).
```

```
reduce(translate(A,B,C),[v(D,E)--v(F,G)|translate(H,B,C)]) :-
```

```
    reduce(A,[H|G]), reduce(I,J--K), reduce(J,v(L,M)),
```

```
    reduce(K,v(N,O)), D is L+B, E is M+C,
```

```
    F is N+B, G is O+C.
```

```
reduce(scale(A,B,C),[]) :- reduce(A,[]).
```

```
reduce(scale(A,B,C),[v(D,E)--v(F,G)|scale(H,B,C)]) :-
```

```
    reduce(A,[H|G]), reduce(I,J--K), reduce(J,v(L,M)),
```

```
    reduce(K,v(N,O)), D is L*B, E is M*C, F is N*B, G is O*C.
```

```
reduce(true,true).
```

```
reduce(false,false).
```

```
reduce([],[]).
```

```
reduce([A|B],[A|B]).
```

```

reduce(A--B,A--B).
reduce(v(A,B),v(A,B)).

```

```

reduce(p,A) :- p=>B, reduce(B,A).
reduce(q,A) :- q=>B, reduce(B,A).
reduce(r,A) :- r=>B, reduce(B,A).
reduce(s,A) :- s=>B, reduce(B,A).
reduce(t,A) :- t=>B, reduce(B,A).
reduce(block1(A),B) :- block1(A)=>C, reduce(C,B).
reduce(block2(A),B) :- block2(A)=>C, reduce(C,B).
reduce(cycle(A),B) :- cycle(A)=>C, reduce(C,B).
reduce(rot(A),B) :- rot(A)=>C, reduce(C,B).
reduce(rot_neg(A),B) :- rot_neg(A)=>C, reduce(C,B).
reduce(rot_pos(A),B) :- rot_pos(A)=>C, reduce(C,B).
reduce(above1(A,B),C) :- above1(A,B)=>D, reduce(D,C).
reduce(alt_rows(A,B),C) :- alt_rows(A,B)=>D, reduce(D,C).
reduce(beside1(A,B),C) :- beside1(A,B)=>D, reduce(D,C).
reduce(mosaic(A,B),C) :- mosaic(A,B)=>D, reduce(D,C).
reduce(row(A,B),C) :- row(A,B)=>D, reduce(D,C).
reduce(union(A,B),C) :- reduce(A,D), union(D,B)=>E, reduce(E,C).
reduce(quartet(A,B,C,D),E) :- quartet(A,B,C,D)=>F, reduce(F,E).

```

```

union([],A)=>A.
union([A|B],C)=>[A|union(C,B)].
rot_pos(A)=>translate(rotate(translate(A,-72,-72),-1.57),72,0).
rot_neg(A)=>translate(rotate(translate(A,-72,-72),1.57),0,72).
block1(A)=>union(A,translate(rot_neg(A),72,0)).
block2(A)=>union(rot_pos(A),
    translate(rotate(translate(rot_pos(A),-72,0),-1.57),72,0)).
row(A,B)=>union(translate(A,C,0),
    union(translate(A,D,0),row(A,E))) :-
    C is 144*B, D is -144*B, E is B+1.

```

```

alt_rows(A,B)=>union(translate(A,0,C),union(translate(A,0,D),alt_rows(A,E))):-
    C is 144*B, D is -144*B, E is B+1.
mosaic(A,B)=>union(alt_rows(row(A,0),0),
    translate(alt_rows(row(B,0),0),0,72)).
beside(A,B)=>union(A,translate(B,36,0)).
above(A,B)=>union(A,translate(B,0,36)).
quartet(A,B,C,D)=>above(beside(C,D),beside(A,B)).
rot(A)=>rotate(A,1.57).
cycle(A)=>union(A,union(rot(A),union(rot(rot(A)),rot(rot(rot(A)))))).
t=>quartet(p,q,r,s).
p=>[..].
q=>[..].
r=>[..].
s=>[..].

```

VI.8 FIRST TWO ELEMENTS OF A LIST

Tamaki [1984] shows how to compile restricted equality theories, expressed by a reducibility predicate \Rightarrow , into Horn clauses with a small search space. For example, the clauses:

```

int(N)=>[N|int(N+1)].
first2([X,Y|Z],[X,Y]).

```

are compiled into:

```

X=>X
int(N)=>[N|Z]:-int(N+1)=>Z
first2([X,Y|Z],[X,Y])

```

Now the query $\text{int}(1) \Rightarrow X, \text{first2}(X, Y)$ succeeds with answer substitution $X = [1, 2 | \text{int}(3)]$, $Y = [1, 2]$. However, it succeeds with infinitely many other distinct answer substitutions, e.g. $X = [1, 2, 3 | \text{int}(4)]$, $Y = [1, 2]$, $X = [1, 2, 3, 4 | \text{int}(5)]$, $Y = [1, 2]$, In LOG(F), however, we would first define:

```

int(N)=>[N|int(N+1)]
first2([U|V])=>g(U,V)
g(U,[A|B])=>[U,A]

```

These would be compiled to:

```

reduce([],[]).
reduce([U|V],[U|V]).
reduce(int(N),Z):-N1 is N+1,reduce([N|int(N1)],Z).
reduce(first2(X),Z):-reduce(X,[U|V]), reduce(g(U,V),Z).
reduce(g(C,X),Z):-reduce(X,[A|B]),reduce([C,A],Z).

```

Now the query `reduce(first2(int(1)),Z)`, would succeed with exactly one answer substitution, `Z=[1,2]`. This is as desired.

VI.9 COMPARING LOG(F) PERFORMANCE WITH THAT OF PROLOG

Programs of similar length, and intellectual complexity are written in both F* and in Prolog. The former were compiled into Prolog, and optimized, before being compared with the latter. The performance figures in Quintus Prolog on a SUN 3/50 are listed in the table below.

For problems in which data structures are always completely evaluated, lazy evaluation cannot reduce lengths of computation. Such problems include list reversal, or sorting. For these, LOG(F) is, on an average, five times slower than Prolog. However, the slowdown for a given problem appears to stay the same, regardless of the size of the input.

For problems in which data structures need only be partially evaluated, e.g. the N-queens problem, or tiling an infinite plane, lazy evaluation can reduce lengths of computation. For these, LOG(F) can be faster than Prolog by factors which are unbounded, i.e. grow with input size, and by factors which are infinite.

Time in milliseconds			
	Prolog	LOG(F)	Prolog/LOG(F)
Reverse: 3200 elements	83	883	0.09
Reverse: 6400 elements	150	1755	0.08
Quicksort: 60 elements	83	539	0.15
Quicksort: 120 elements	250	1261	0.19
Sieve: First 50 primes	422	1261	0.33
Sieve: First 100 primes	1816	4511	0.40
All permutations of [1,2,3,4,5]	427	516	0.82
All permutations of [1,2,3,4,5,6]	3000	3172	0.94
8-Queens: All solutions	62783	17511	3.58
9-Queens: All solutions	635144	86539	7.33
15-Queens: First solution	>30 minutes	30300	>60
Infinite plane: First vector	∞	~ 0	∞

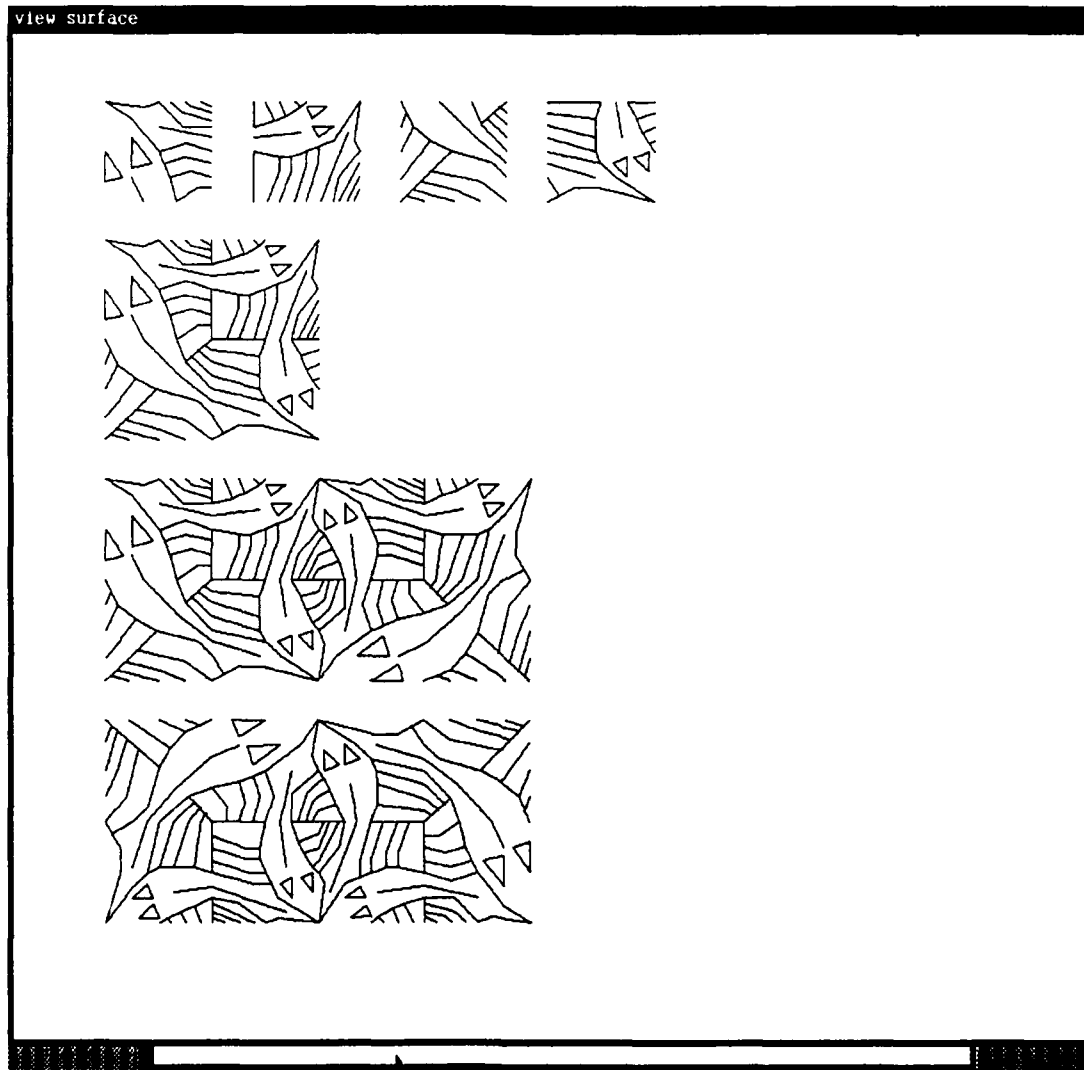


Figure 1. Some Graphical Primitives

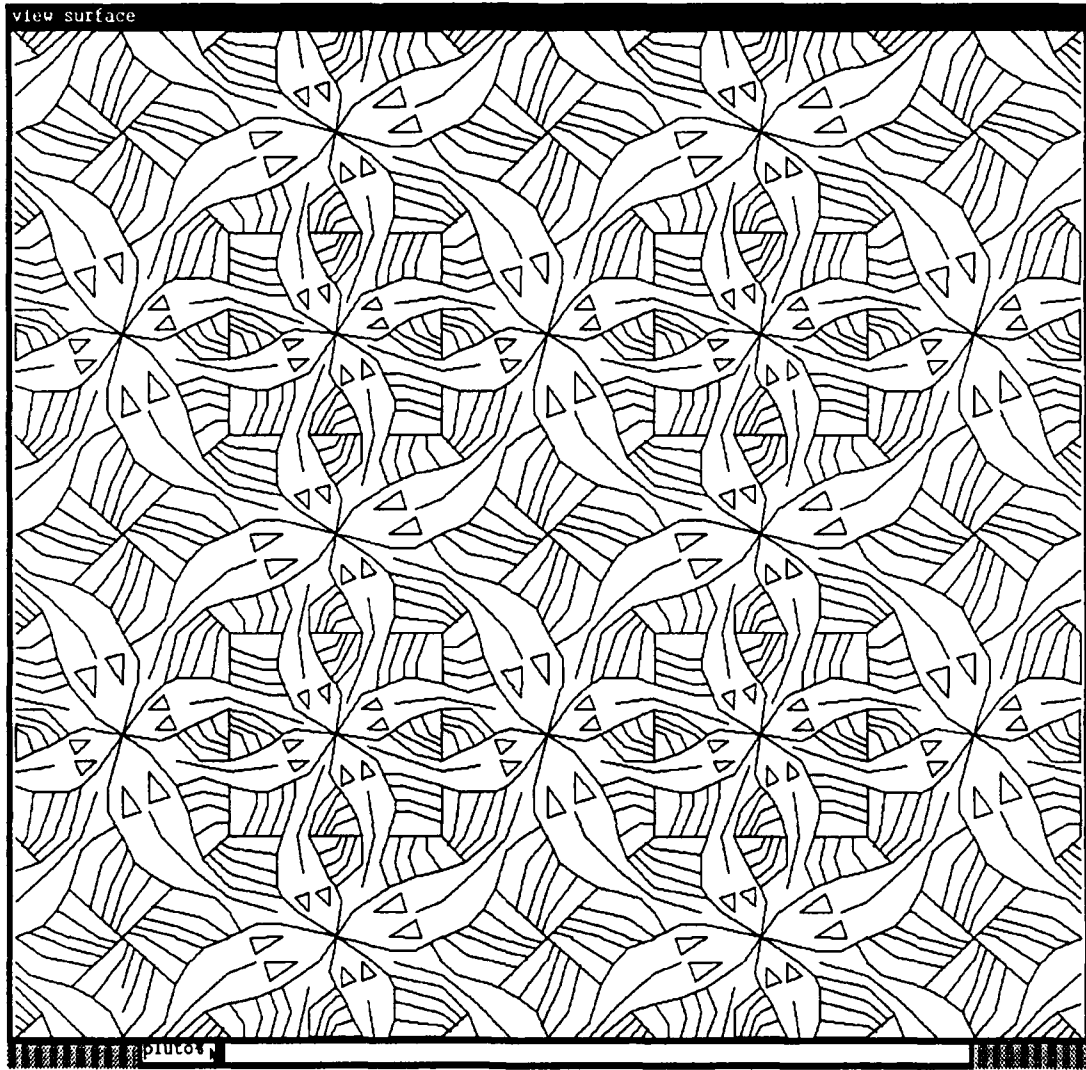


Figure 2. Square Unlimit

VII. SUMMARY AND CONCLUSIONS

A new approach for combining logic programming, rewriting, and lazy evaluation is described. It rests upon *subsuming* within logic programming, instead of upon *extending* it with, rewriting, and lazy evaluation.

F^* is a non-terminating, non-deterministic rewrite rule system. The reduction strategy for it, *select*, is reduction-complete. DF^* is a subset of F^* , and is also non-terminating. DF^* satisfies confluence, directedness, and minimality. Reduction-completeness, and minimality enable *select* to exhibit, respectively, weak and strong forms of laziness.

F^* can be compiled into Horn clauses in such a way that when SLD-resolution interprets them, it directly simulates the behavior of *select*. In particular, it is made to exhibit laziness. $LOG(F)$ is defined to be a logic programming system augmented with an F^* compiler, and the equality axiom $X=X$. Since clauses obtained by compiling F^* programs can be called from other logic programs, $LOG(F)$ is proposed as a combination of logic programming, rewriting, and lazy evaluation.

$LOG(F)$ offers, perhaps for the first time, an efficient implementation of lazy evaluation within a widely used language, namely, Prolog. For problems in which lazy evaluation cannot reduce lengths of computation, $LOG(F)$ is somewhat slower than Prolog. For problems in which lazy evaluation does reduce lengths of computation, $LOG(F)$ can be faster than Prolog by factors which are unbounded, i.e. grow with input size, and factors which are infinite.

$LOG(F)$ can also be used to implement useful cases of the rule of substitution of equals for equals. Confluence of DF^* yields a new proof of the confluence of combinatory logic. Finally, DF^* seems to be a good candidate for implementation on parallel machines. It seems to offer a reasonable compromise between sequential execution and unbounded parallelism. Due to directedness of DF^* , arguments of f in $f(t_1, \dots, t_m)$ can be simplified in parallel, however, they would be simplified lazily.

VIII. ACKNOWLEDGEMENTS

I am very grateful to my advisor, Professor D.S. Parker, for his guidance, encouragement, and criticism throughout the development of the thesis on which this paper is based. I also thank Professors Y.N. Moschovakis, David Jefferson, and Milos Ercegovic, and Dr. Hassan Ait-Kaci for their many valuable suggestions.

IX. REFERENCES

- Abramson, H. [1986]. A prolog definition of HASL, a purely functional language with unification-based conditional binding expressions. In *Logic programming: functions, relations and equations* (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, N.J.
- Ait-Kaci, H., Nasr, R. [1986]. Residuation: A paradigm for integrating logic and functional programming. MCC Technical Report AI-359-86, Austin, TX.
- Ait-Kaci, H., Lincoln, P., Nasr, R. [1987]. Le Fun: Logic, Equations and Functions. *Proceedings of symposium on logic programming*, San Francisco.
- Apt, K.R., van Emden, M.H. [1982]. Contributions to the theory of logic programming. *Journal of the Association for Computing Machinery* vol. 29, no. 3, July 1982.
- Barendregt, H.P. [1977]. The type free lambda-calculus, in: *Handbook of Mathematical Logic*, ed. John Barwise, North Holland Publishing Company.
- Barbuti, R., Bellia, M., Levi, G. [1986]. LEAF: A language which integrates logic, equations, and functions. In *Logic programming: functions, relations and equations* (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, N.J.
- Barrow, H. [1983]. Proving the Correctness of Digital Hardware Designs. *Proceedings of the National Conference on Artificial Intelligence*, Washington, D.C.
- Bellia, M., Levi, G. [1986]. The relation between logic and functional languages: A survey. *Journal of Logic Programming*, October.
- Berry, G., Levy, J.-J. [1979]. Minimal and optimal computations of recursive programs. *Journal of the Association for Computing Machinery*, vol. 26, no. 1, pp. 148-175.
- Bundy, A., Welham, W. [1981]. Using meta-level inference for selective application of multiple rewrite rule sets in algebraic manipulation. *Artificial Intelligence*, vol 16, no. 2, May.
- Burstall, R.M., MacQueen, D.B., Sannella, D.T. [1980]. HOPE: An experimental applicative language. *Proceedings of 1980 Lisp Conference*. Stanford, California.
- Burstall, R., Darlington, J. [1977]. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, vol. 24, No. 1.

- Church, A. [1941]. *The calculi of lambda-conversion*. Annals of mathematical studies, number 6. Princeton University Press, Princeton.
- Clark, K.L., McCabe F. [1979]. Programmer's guide to IC-Prolog. *CCD Report 79/7*, London: Imperial College, University of London.
- Clark, K.L. [1980]. Predicate Logic as a computational formalism. Research monograph, Imperial college, University of London.
- Clark, K.L., McCabe, F. [1982]. PROLOG: A Language for implementing expert systems. *Machine Intelligence 10*, (eds.) J. Hayes and D.J. Michie.
- Clark, K.L., Gregory, S. [1986]. PARLOG: Parallel programming in logic. *ACM transactions on programming languages and systems*, 8,1.
- Curry, H.B., Feys, R. [1958]. *Combinatory Logic, vol I*, North Holland, Amsterdam.
- Darlington, J., Field, A.J., Pull, H. [1986]. The unification of functional and logic languages. *Logic programming: functions, relations and equations* (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, New Jersey.
- Davis, M., Putnam, H. [1960]. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7, pp. 201-215.
- DeGroot, D., Lindstrom, G. (editors) [1986]. *Logic programming. Functions, relations and equations*. Prentice Hall, N.J.
- Dershowitz, N., Josephson, N.A. [1984]. Logic Programming by completion. *Proceedings of second international logic programming conference*, Uppsala University, Sweden.
- Digricoli, V.J., Harrison, M.C. [1986]. Equality-based binary resolution. *Journal of the Association for Computing Machinery*, vol. 33, no. 2, April.
- Dijkstra, E. [1976]. *A discipline of programming*. Prentice-Hall, Englewoods Cliffs, N.J.
- Dincbas, M., van Hentenryck, P. [1987]. Extended unification algorithms for the integration of functional and logic languages. *Journal of logic programming*, vol. 4, No. 3.
- Fay, M. [1979]. First order unification in an equational theory. *Proceedings of the 4th*

conference on automated deduction.

Frege, G. [1879]. Begriffsschrift. A formula language, modelled upon that of arithmetic, for pure thought. In *From Frege to Goedel: A source book in mathematical logic, 1879-1931*. Harvard University Press, Cambridge, MA.

Fribourg, L. [1984]. Oriented equational clauses as a programming language. *Journal of Logic Programming* vol 1, pp. 165-177.

Friedman, D.P., Wise, D.S. [1976]. CONS should not evaluate its arguments, in *Automata, Languages and Programming*, eds. S. Michaelson, R. Milner, Edinburgh University Press, Edinburgh.

Gallagher J. [1982]. Simulating Corouting for the 8 Queens Problem. *Logic Programming Newsletter* 3, Summer 1982.

Gallaire, H., Lasserre, C. [1982]. Metalevel Control for Logic Programs, *Logic Programming* eds. K. Clark, S.-A. Tarlund, Academic Press, New York.

Gallaire, H., Minker, J. (editors) [1978]. *Logic and Databases*, Plenum Press, New York.

Gallier, J.H., Raatz, S. [1986]. SLD-resolution methods for Horn clauses with equality based on E-unification. *Proceedings of 1986 symposium on logic programming*, Salt Lake City, Utah.

Goguen, J.A., Meseguer, J. [1986]. Equality, types and generic modules for logic programming. *Logic programming: functions, relations and equations* (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, N.J.

Green, C.C. [1969]. Theorem proving by resolution as a basis for question-answering systems. *Machine Intelligence*, 4, Edinburgh University Press, pp 183-205.

Greene, K.J. [1985]. A fully lazy higher order purely functional programming language with reduction semantics. CASE Center technical report no. 8503, Syracuse University, N.Y.

Hansson, A., Haridi, S., Tarlund, S.-A. [1982]. Properties of a logic programming language, in: *Logic Programming* eds. K. Clark, S.-A. Tarlund, Academic Press, New York.

Henderson, P. [1980]. *Functional Programming: Application and Implementation*.

Prentice Hall International, N.J.

Henderson, P. [1982]. Purely functional operating systems. In *Functional programming and its applications. An advanced course*. (eds.) J. Darlington, P. Henderson, D.A. Turner.

Henderson, P. [1982]. Functional Geometry. *Proceedings of the ACM Symposium on Lisp and Functional Programming*. Pittsburgh, PA.

Hill, R. [1974]. LUSH Resolution and its completeness. DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh.

Hoffman, C.M., O'Donnell, M.J. [1982]. Programming with equations. *ACM Transactions on programming languages and systems*. January.

Hoelldobler, S. [1987]. Equational logic programming. *Proceedings of Fourth Symposium on logic programming*, San Francisco, CA.

Hopcroft, J., Ullman, J. [1979]. *Introduction to automata theory, languages and computation*. Addison Wesley, Menlo Park, CA.

Huet, G. [1980]. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the Association for Computing Machinery*, October.

Huet, G., Levy, J.-J. [1979]. Call by need computations in non-ambiguous linear term rewriting systems. IRIA technical report 359.

Huet, G. [1975]. A unification algorithm for typed λ -calculus. *Theoretical computer science* 1 (1975) 27-57.

Hullot, J.-M. [1980]. Canonical forms and unification. *Proceedings of 5th conference on automated deduction, Lecture Notes in Computer Science*, 87, Springer Verlag.

Jaffar, J., Lassez, J.-L., Maher, M.J. [1984]. A theory of complete logic programs with equality. *Journal of logic programming*, vol. 1, no. 3.

Kahn, K. [1986]. Uniform -- A language based upon unification which unifies (much of) Lisp, Prolog, and Act 1. In *Logic programming: functions, relations and equations* (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, N.J.

Kahn, G., MacQueen, D. [1977]. Coroutines and Networks of Parallel Processes.

Information Processing-77, North-Holland, Amsterdam.

Knuth, D.E., Bendix, P.B. [1970]. Simple word problems in universal algebras. *Computational problems in abstract algebra*, ed. J. Leech, Pergamon.

Kohavi, Z. [1978]. *Switching and finite automata theory*. McGraw Hill, New York.

Komorowski, H.J. [1982]. QLOG - The programming environment for Prolog in Lisp. *Logic Programming* eds. K. Clark, S.-A. Tarnlund, Academic Press, New York.

Kornfeld, W. [1983]. Equality for Prolog. *Proceedings of International Joint Conference on Artificial Intelligence*. Karlsruhe, West Germany.

Kowalski, R. [1979]. *Logic for Problem Solving*, Elsevier North Holland, New York.

Lloyd, J. [1984]. *Foundations of logic programming*. Springer Verlag, New York.

Malachi, Y., Manna, Z. [1986]. Tablog: A new approach to logic programming. In *Logic programming: functions, relations and equations* (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, N.J.

McCarthy, J. [1960]. Recursive functions of symbolic expressions and their computation by machine. *Communications of the Association for Computing Machinery*, 3, pp. 184-195.

Miller, D.A., Nadathur, G. [1986]. Higher-order logic programming. *Proceedings of third international conference on logic programming*. Lecture notes in computer science 225, (ed.) E. Shapiro, Springer Verlag, New York.

Narain, S. [1985]. A technique for doing lazy evaluation in logic. *Proceedings of IEEE Logic Programming Symposium*, Boston, MA.

Narain, S. [1986]. MYCIN: The expert system and its implementation in LOGLISP. In *Logic programming and its applications*, eds. D.H.D. Warren, M. van Caneghem, Ablex publishing company, N.J.

Narain, S. [1986]. A Technique for Doing Lazy Evaluation in Logic. *Journal of Logic Programming*, vol. 3, no. 3, October.

Narain, S. [1988]. LOG(F): An optimal combination of logic programming, rewriting and lazy evaluation. Ph.D. Thesis, Department of Computer Science, University of

California, Los Angeles.

O'Donnell, M.J. [1985]. *Equational logic as a programming language*. MIT Press, Cambridge, MA.

Pereira, F.C.N., Warren, D.H.D. [1980]. Definite clause grammars for natural language analysis. A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence Journal*, 13, pp. 231-278.

Pingali, K., Arvind. [1985]. Efficient demand-driven evaluation. Part 1. *ACM transactions on programming languages and systems*, April 1982.

Rabin, M.O. Theoretical impediments to artificial intelligence.

Reddy, U.S. [1985]. Narrowing as the operational semantics of functional languages. *Proceedings of the 1985 symposium on logic programming*, Boston.

Robinson, G., Wos, L. [1969]. Paramodulation and theorem proving in first order theories with equality. *Machine Intelligence 4*, (eds.) B. Meltzer, D. Michie, M. Swann.

Robinson, J.A. [1965]. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12, pp 23-41.

Robinson, J.A. [1979]. *Logic: Form and Function. The Mechanization of Deductive Reasoning*. Elsevier North Holland, New York.

Robinson, J.A., Sibert, E.E. [1982]. *LOGLISP: Motivation, Design and Implementation. Logic Programming* eds. K. Clark, S.-A. Tarnlund, Academic Press, New York.

Robinson, J.A. [1984]. Editor's introduction. *Journal of logic programming*, vol. 1, no. 1, June.

Robinson, J.A. [1987]. Beyond LOGLISP: Combining functional and relational programming in a reduction setting. *Machine Intelligence 11*.

Robinson, J.A., Greene, K.J. [1987]. *New Generation Knowledge Processing*, vol. III. RADC-TR-87-165. Rome Air Development Center, Griffis Air Force Base, NY 13441.

Rosser, J.B. [1982]. Highlights of the history of the lambda-calculus. *Proceedings of the ACM Symposium on Lisp and Functional Programming*. Pittsburgh, PA.

- Sato, M., Sakurai, T. [1986]. QUTE: A functional language based on unification. In *Logic programming: functions, relations and equations* (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, N.J.
- Shapiro, E. [1983]. A subset of Concurrent Prolog and its interpreter. *ICOT technical report TR-003*, February 1983.
- Shapiro, E., Takeuchi, A. [1983]. Object-oriented Programming in Concurrent Prolog. *New Generation Computing I* (1983), OHMSA LTD and Springer Verlag, Japan.
- Subrahmanyam, P.A. and You J.-H. [1984]. Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming. *Proceedings of IEEE Logic Programming Symposium*, Atlantic City, N.J.
- Tamaki, H. [1984]. Semantics of a logic programming language with a reducibility predicate. *Proceedings of IEEE Logic Programming Symposium*, Atlantic City, N.J.
- Turner, D. [1979]. A New Implementation Technique for Applicative Languages, *Software Practice and Experience*, 9, pp. 31-49.
- Ueda, K. [1986]. Guarded Horn Clauses. Ph.D. Thesis. University of Tokyo. Tokyo, Japan.
- van Emden, M.H., Yukawa, K. [1987]. Logic programming with equations. *Journal of Logic Programming*, vol. 4, no. 4.
- Vuillemin, J. [1974]. Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9, pp. 332-354.
- Wadsworth, C.P. [1976]. The relation between computational and denotational properties for Scott's D_{∞} -models of the lambda-calculus. *SIAM Journal of Computing*, vol. 5, no. 3, September.
- Warren, D.H.D., van Caneghem, M. (editors) [1986]. *Logic Programming and its Applications*, Ablex Publishing, N.J.
- Warren, D.H.D., Pereira, L.M., Pereira, F.C.N. [1977]. Prolog - the language and its implementation compared with Lisp. *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, SIGPLAN Notices 12, No. 8, and SIGART Newsletter 64, pp 109-115.

Yamamoto, A. [1987]. A theoretical combination of SLD-resolution and narrowing. *Proceedings of fourth international conference on logic programming*, Melbourne, Australia.